ANE v 0.8

Mathematica Setup (not relevant if you're reading the PDF)

Need to use a conditional on "printing to PDF" to hide this!

This writeup was all done in Mathematica. If you have access to Mathematica, you can download the companion notebook and look at the actual numbers, draw your own graphs from those numbers etc. But most people don't have Mathematica, so for you I've printed the notebook to a PDF.

If you use Mathematica, we need a way to paste results data from the command line apps into Mathematica.
Easiest solution appears to be
http://szhorvat.net/pelican/pasting-tabular-data-from-the-web.html

When you first open this notebook, say yes to "Allowing Dynamic Content". You will need this to activate the two UI elements ("Show Input" and "Outline") at the top of the document.
Next choose Evaluate Initialization Cells from the Evaluate menu (this will take a few tens of seconds to execute). This will load all internal variables (specifically all the many arrays of measurement data) into Mathematica, allowing you, if you want, to plot the graphs in different ways, or otherwise interact with the data.

Note the "Show Input" button at the top of this notebook; toggle it if you want to see the (sometimes copious!) input data for any particular graph.
The Mathematica code below adds that functionality to this notebook (not shown when "Show Input" is untoggled.

Also remember ctrl-clicking on a graph brings up a contextual menu, one of whose items, "Get Coordinates" is often useful in getting a quick, reasonably accurate feel for the coordinates of a point .

Introduction

If you want a very simple intro to the ANE (what devices have what version, how to steer your code to it, things like that) look here
https://github.com/hollance/neural-engine?tab=readme-ov-file .
If you want to understand much more of how the ANE hardware works, keep reading.

The Language of Neural Networks

Understanding neural processors is even more fraught than understanding GPUs. The designs are much newer, and there isn't yet a pool of papers and background knowledge the way there is for nVidia or AMD, knowledge that while not perfectly applicable to other designs, at least allows one to understand the issues. But we will do our best!

First let's start by pointing out terminology problems. Neural networks suffers from the sort of terminology slip that besets any rapidly changing field. Words or phrases begin with more or less sensible referents, the referents change but it's easiest to retain the word, and so after a decade or two the word has no obvious connection to how it is used.
Some examples are:
- tensor. To a mathematician a tensor is an element of a vector space whose numerical representation changes in a particular way under change of basis. To a physicist all that is true, plus an expectation of certain more "geometric" type properties.
But to a neural network a "tensor" is **just** an array of numbers, possibly a 1D array, possibly a 3D array, possibly a 6D array. There is, most importantly, no expectation that change of basis and mixing together of the numbers necessarily makes sense; it may or may not on a case by case basis. A 3D table of say (weight, height, wealth) measurements would be acceptable to a neural net as a "tensor" in a way that would horrify a mathematician or physicist.
However slices of the tensor may be legitimate vectors, with legitimate vector operations acceptable; for example a plane of luminance image data can be seen as a legitimate tensor and various operation, both obvious (linear filtering via convolution) or less obvious (decomposition into image basis vectors, eg Karhunen–Loève) may be legitimate.
Bottom line is you have to constantly maintain your guard. You cannot blindly throw theorems from linear algebra at these "tensors" but you certainly may be able to for some use cases or for slices of the tensor.

- I have referred to 1D, 2D, or 6D arrays. This can be confusing in that reference to a 3D array or 3D vector may make you think of a set of 3 numbers, or it may make you think of a volume of numbers with a width, depth, and height... For this reason is it more desirable to talk of a *rank 3* array or tensor. The shape of a tensor is both its rank and the dimensions of each element of the rank (eg width, height, depth of a rank 3 tensor). Again mathematically there are only a few *legitimate* ways to reshape a tensor (ie ways that generate results independent of the representational basis) but in the world of "tensors" as simple arrays, all manner of reshaping is performed, more or less on the basis of "does it appear to work".

- once again to a mathematician or even an EE "filtering" means linear filtering, which means a very particular type of operation, implemented as a convolution. For neural nets filtering is somewhat vaguer. It frequently does refer to a linear filtering as implemented by convolution, but it may refer to

alternative “filtering-like” operations, where you take some small portion of the signal, generate some representative value (eg maximum), then slide your window sideways and repeat.

Next I’m going to assume you know what convolutions are; if you don’t you better go out and learn! Convolutions can (you learned in one of your calculus classes) be implemented via Fourier Transforms; but this is impractical for the short (3 × 3 or 5 × 5 convolutions used in neural nets).
For these short convolutions, you can implement what is known as Winograd Convolution, which is one of these things, like Strassen Matrix Multiplication, where you can re-arrange the computation to reduce the number of multiplies at the expense of an irregular data access pattern. Sounds cool; it’s not clear that it’s useful for hardware (though it may be useful for CPU implementations, where people are experimenting and constantly changing code, and want the CPU for debuggability). But if you only have matrix multiply hardware, then ...

This will establish an on-going theme in neural compute. There are certain optimal ways on doing things on *particular* hardware (*cough* nVidia *cough*) that are not universal to all hardware, but which get immense attention because of nVidia’s (well-deserved, to be fair) central place in the neural network hardware universe. Don’t be narrow-minded! It’s valuable to know why things are done a certain way on nV hardware, but it’s as important to understand the primary operation, and how it can be executed differently on alternative hardware.

For the most part, when you see Winograd Convolution just treat it as meaning “perform a convolution, more or less directly rather than via FFT)”.

The fixed values used in executing a neural network are usually called *weights*. In the context of convolution, you might think of these as filter values or kernels.

The non-fixed values used in executing a neural network (ie the particular image being recognized, or the particular text prompt being interpreted) I am calling *signal*, which is a non-standard term, but I think best conveys the idea – signal flows into each layer of a neural network, is modified by weights, and flows out to act as signal for the next layer of the network.

The more standard term of “signal” is *activations*, and you will see this term a lot. Another term for the same concept, used less frequently and in slightly more specialized areas, is *output fmap* (which stands for *output feature map*), and tends to refer to the signal values after they have passed through the final processing of a given layer, which may be some combination of ReLU (ie delinearization function), some sort of normalization, and maybe some sort of amplifying the largest values, something like SoftMax.

Next hardware to accelerate neural nets is still the wild west. There are some basic operation that are required and which take up most of the compute (we’ll describe these soon) but the optimal way to balance performance, power, area, flexibility, etc is much like GPUs in the late 1990s; there’s no clearly optimal solution.
For example in convolving 2D signals (eg image recognition) do you want your hardware laid out in 2D, which filter coefficients locked into place and the image moving sideways from one MAC to the next each cycle? This is, more or less, Google. Power efficient, but of limited flexibility if you want to switch to language, or audio, or anything not 2D.

Or do you want to represent the convolution via a “flattened, unrolled” matrix and run it through a matrix multiplier? This is more or less nVidia. Reflects the fact that they have matrix multiplies and no more specialized hardware, but it’s not optimal in any way. Not optimally flexible, not area efficient, horribly memory and bandwidth inefficient. But that same hardware is a lot more flexible for non image convolution tasks.
Or do you want to provide fairly generic 1D hardware (something like very long SIMD registers) along with some clever addressing/routing hardware that allow you to turn 2D convolution (and most other compute intensive neural tasks) into essentially a super-SIMD problem? That’s more or less Apple, and we’ll see that it strikes a very nice balance of all three – power, area, and flexibility

Common Layers of Neural Networks

So, OK, now that we know the above, what is a neural network accelerator?
I’m not going to describe much what neural networks are or how/why they work, I leave that to you.
But an NN looks something like

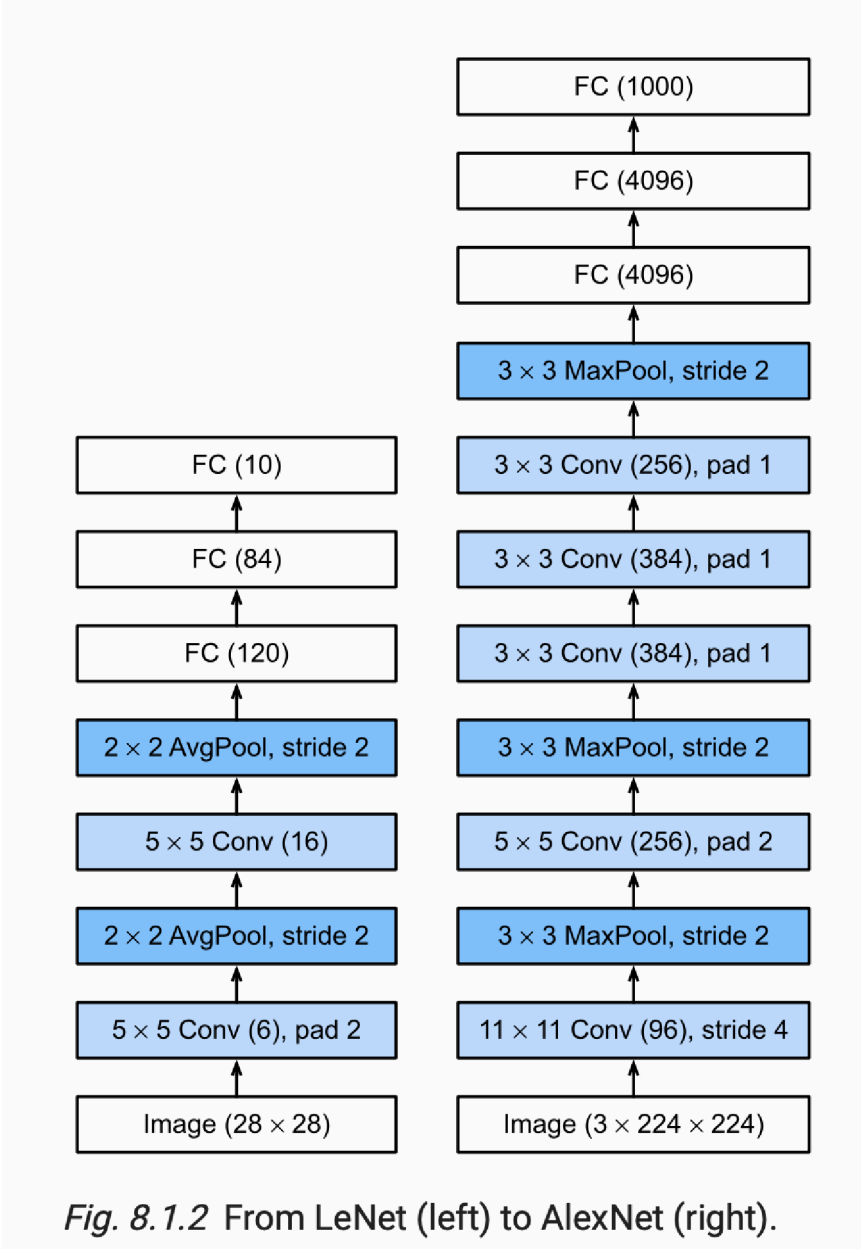


Fig. 8.1.2 From LeNet (left) to AlexNet (right).

(from http://d2l.ai/chapter_convolutional-modern/alexnet.html , a site you might want to read in detail, from the start, if most of this is new to you).

So let's note some important issues.

First the NN is described as a sequence of "abstract" operations, abstract meaning that any particular task (like 11x11 convolution) implies a large number of operations (eg multiply-add's) but in a very structured way. You could turn this into nested loops on a CPU or slightly less nested loops on a GPU, but optimal would be to inform hardware of the overall task and have it do the work as the result of one "instruction".

One version of code implementing the above is

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

and you might want your NN accelerator, as much as possible, to have more or less a one-to-one correspondence between each instruction and each of the function calls above.

This is essentially what we will see with the ANE. You give the ANE “tasks” which are the equivalent of GPU kernels. Tasks are made up of a “task list”, and the elements of the task list are, approximately, interleavings of an instruction that describes a large data movement and an instruction that describes and entire network layer (convolution, pooling, ReLU, whatever). The way this is possible is that the ANE has a large number of distributed “loop managers” with names like “rasterizer” or “shifter”. These are configured by the instruction and then operate over many cycles moving data as appropriate. You can do the equivalent of, for example, setting up a depth-3 nested loop then have it run autonomously without needing any communication with the rest of the system.

So what are these layers? Some common tasks include
- convolution. In the vision case, you should be aware of how this maps onto finding lines/edges in small patches of the image.
- pooling. This is shrinking the image in some way, for example consider 2×2 patches and finding the maximum value in that patch
- local normalization. This again considers small patches, and finds the largest value, but then scales all values to the largest value, so the range of values in from -1 to1.
- a variant of this is normalizing the entire signal (eg image) by calculating the mean and standard deviation, then subtracting the mean and scaling by the standard deviation
- sometimes we more or less want to clip calculated values into “feature present or not present”, done by mapping output values through a nonlinear function, eg ReLU or arctan
- fully connected layers which are just large matrix-vector multiplies.

If you look at both the LeNet and AlexNet diagrams, you will see versions of these. Essentially (engage vigorous hand waving) the first layer finds edges, so that we can associate with each small patch of the image a vector which says “strong presence of edges of type 1, 3, 27, and 99” (each edge type may be, for example, at a different orientation). The next layer then, for each small image patch, takes in

the set of edge types and from that generates a set of texture types. We now get a representation of each patch as “having a lot of textures 2, 6, 28 and 147”.
Generically we can call this “extraction of feature vectors”, where we convert the image as pixels into successively more abstract (and less localized, less dependent on precise image location, scale, orientation, etc) features.
Finally the FC (Fully Connected) layers are the ones that basically say that an image that has a large amount of edges of type 1, 3, 27 and textures 2, 6, 28 and 147 in multiple patches maps onto a cat – or whatever.

So we want hardware that can do all of these – convolution, pooling, non-linear function lookup, dot products, and matrix -vector products. Amusingly (at least for the networks I’ve described) matrix-matrix products not really part of the job description! They do appear in training, because you might, eg, implement multiple different signals as a batch of distinct signals, and then stages that look like matrix(weights) vector (signal) multiply can be coded to look like matrix(weights) vector (multiple signals packed together) multiply.
Whether this is optimal or not depends, of course, on your hardware – when all you have is a matrix-matrix multiplier...

History of the ANE

We now have an idea of what we want to do and that there are many ways of doing it. Apple’s particular solution grows out of their *camera* history, and can only be understood in light of that history.
(As a technical point, the A11 had what Apple called an Apple Neural Engine, possibly based on a design from Lattice Semiconductor. This was very limited in power and abilities; it seems to have been limited to FaceID and Memoji [which require detecting features in a face, and transferring those features to warp a cartoon]. I’m not going to bother with this design because it seems, like the A10 big-little CPU design, to have been a dead end, probably useful for learning and getting started, but not relevant to long term evolution. What I discuss below is the path to the 2nd and later generation ANE’s, but I will refer to the ANE of the A12 as version one, the first *real* ANE.)

(2012) Early ISP processing

We can start with (2012) https://patents.google.com/patent/US9025867B2 *Systems and methods for YCC image processing*.
This is a massive patent and not worth reading (unless early mobile ISPs are really interesting to you) but I want to point out two features
- first it’s doing a lot of stuff, but all hardwired; it’s not really programmable in any sense interesting to us
- fig 116 which shows that we’re doing a whole lot of different operations across the entire image to gather data for optimization of the final photo.

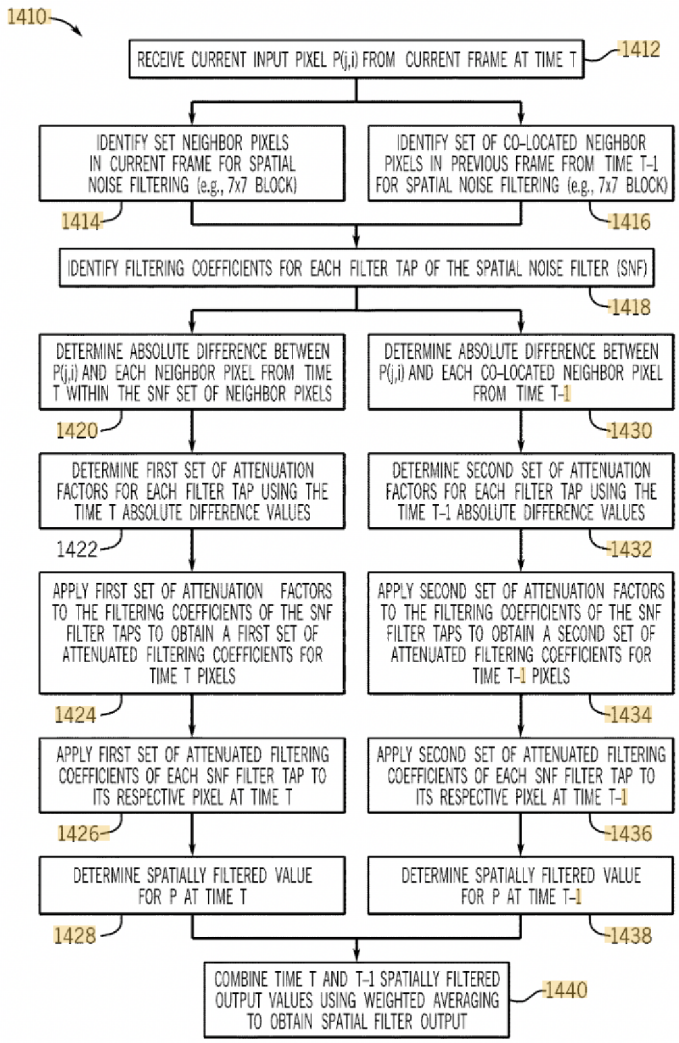


FIG. 116

These operations include

- comparing pixels with their neighbors in space,
- comparing pixels with the same pixel in an earlier frame,
- calculating/modifying filter coefficients (ie not just using hardwired filter coefficients), and
- combining as a weighted average two separately filtered images.

Much of the filter coefficient modifications are driven by earlier steps (which we will not discuss!) which collect data about the image, perform local normalization, and so on.

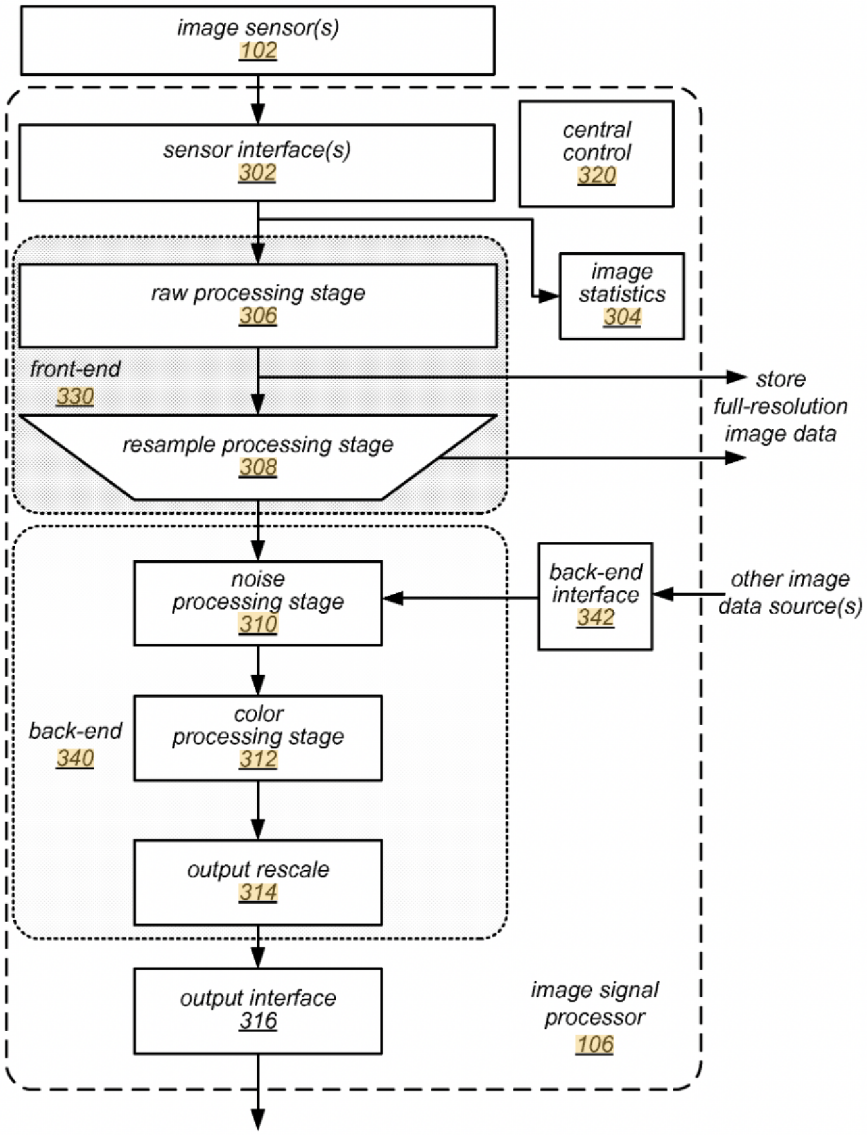
This provides a minimum set of functionality we will want when we make the ISP more programmable, and we can see already see elements (though mostly hardwired) of convolution, pooling, and so on.

If you want to go even further back 2010 https://patents.google.com/patent/US20120081553A1 *Spatial filtering for image signal processing* gives some of the theory of this noise filtering, based on statistics gathered from the image.

(2015) Last stage of ISP before the convolution engine

Apple's camera patents now go through three stages, from fixed hardware through "configurable" to splitting off the ANE which becomes "programmable".

The end stage of fixed hardware is seen in (2015) https://patents.google.com/patent/US10269095B2 *Dynamically determining filtering strength for noise filtering in image processing*. Let's look at one diagram, to compare with what's coming:

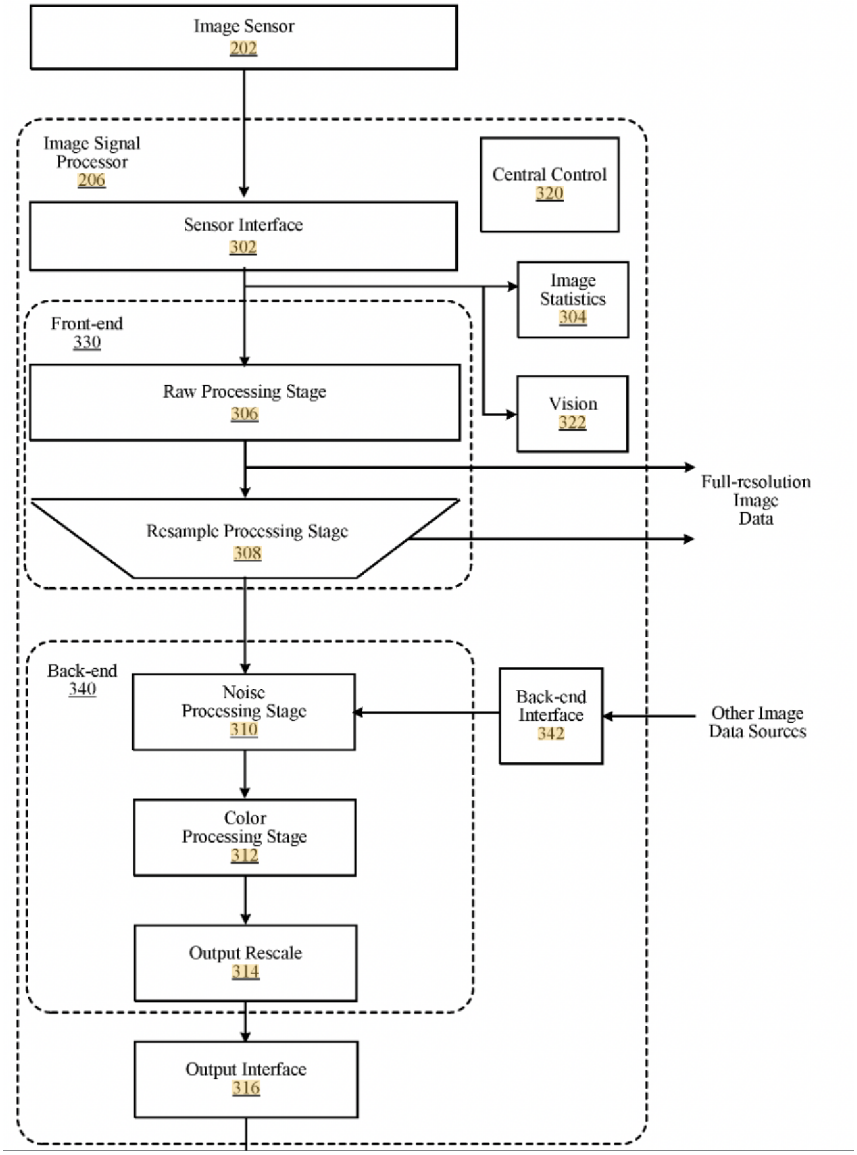


To understand the diagram, appreciate that the front-end is basically *camera-specific*. It's job is to deal with quirks, limitations, and flaws in the front-end to convert from a camera-specific image to an interchangeable image. The back-end's job, in contrast, is to act as a generic ISP, capable to taking images from any source (eg from video playback, or pictures from the internet) and then apply image processing algorithms to them, either as part of camera capture (to make nice looking images in point-and-click mode) or eg when editing photos.

(2016) Convolution engine added to the ISP

A year later we get something that's superficially similar, but starts the "configurable" stage of the evolution.

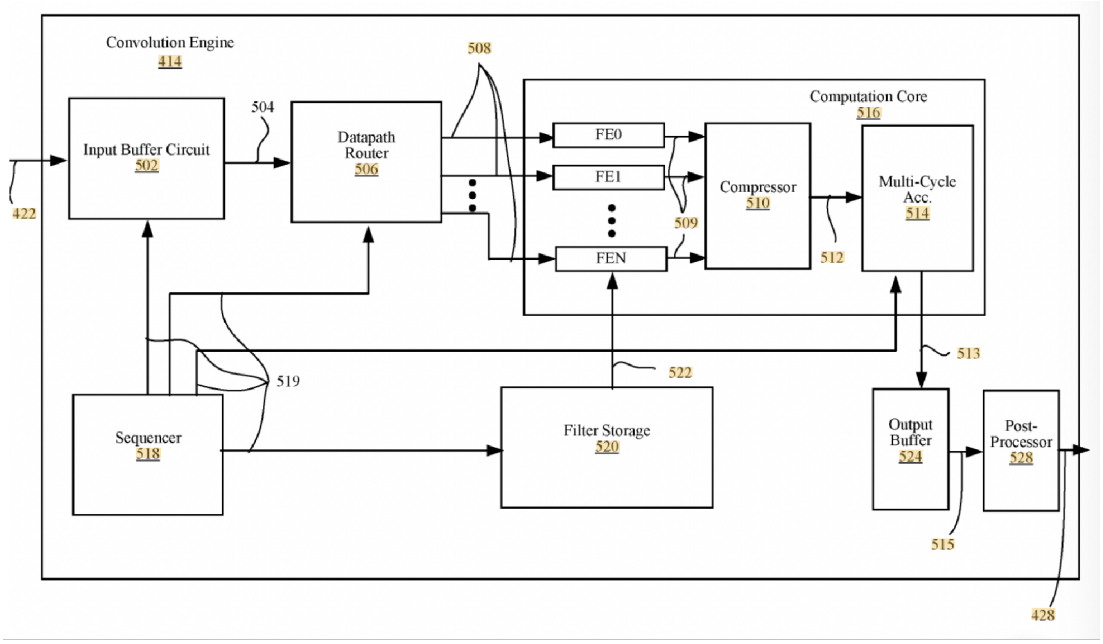
(2016) https://patents.google.com/patent/US9858636B1 *Configurable convolution engine*.



The big deal is that new block called *Vision*.

This has two parts. One, called the HoG (Histogram of Gradients) Engine captures fancy statistics. I think that it, for example and as a very simple case, is what provides the lines showing that you are holding the camera at an angle, not perfectly aligned. These statistics also, for example, help with autofocus or determining exposure length.

But what we care about is the second part, the Convolution Engine. This looks like so:



The ANE starts off as basically a fancy version of this. So it makes sense for us to study it in some detail.

We are already going to start seeing terminological drift and it only get's worse! So just accept that some things are named (or diagrammed) in a way that doesn't completely make sense.

Signal (ie the image, but also accompanied by statistics or HoG data) comes in as 422, and exits at 428.

Importantly output 428 can loop back into 422 so that we can apply multiple successive filters.

Units FE0 to FEN are multiply-add units. The patent never says so, but let's assume there are, say, 16 of them.

Forget Compressor for now.

Multi-Cycle Acc (crazy name that will make ever less sense as this evolves) is basically a small amount of register storage, let's think of it as say eight(?) storage values for each multiply-add units.

So basically the pattern is each cycle

- signal data flows into each FE (different values for each FE)
- filter values flow into each FE (a broadcast, same filter value for each FE)
- the signal is multiplied by the filter value and added to a previous value which is supplied by Multi-Cycle Acc 514

Assume we want to implement a genuine convolution of the simplest real form, and for now let's assume a 1D signal (something like audio, and we are implementing a ridiculously simple 3 tap filter, eg to reduce high frequencies). As I describe things below, I am going to ignore details like edge effects (at the edge do we zero pad? repeat the first/last value of the signal? wrap-around like a circular convolution?) just to keep things simple.

So essentially we want to calculate

$o_0 = k_0\, i_0 + k_1\, i_1 + k_2\, i_2$

$o_1 = k_0\, i_1 + k_1\, i_2 + k_2\, i_3$

$o_2 = k_0\, i_2 + k_1\, i_3 + k_2\, i_4$
and so on. How can we do this, using this hardware, at minimal energy?
The thing you have to remember, in this and every subsequent version, is that we calculate an *n*-tap convolution over *n* cycles.
If you look at the above you will see that we can in the first cycle
- load values $i_0..i_{15}$
- broadcast kernel (ie filter) value k_0 to all the FE's
- multiply each *i* by k_0
Then the next cycle we
- load values $i_1..i_{16}$ (note that each input value is shifted by 1)
- broadcast kernel (ie filter) value k_1 to all the FE's
- multiply each *i* by k_1 and add in the previously calculated *i* times k_0
Repeat again broadcasting k_2
Now we have 16 result (filtered) values that we can send out. We now repeat starting with
$o_{16} = k_0\, i_{16} + k_1\, i_{17} + k_2\, i_{18}$
$o_{17} = k_0\, i_{17} + k_1\, i_{18} + k_2\, i_{19}$
$o_{18} = k_0\, i_{18} + k_1\, i_{19} + k_2\, i_{20}$

So each cycle we figure out which kernel values we need (cycling through the three kernel elements), and which input values go where. Deciding on the input values is not called out in the diagram, but it will become the job of a small piece of logic called the *shifter*, which is, like I said, basically a hardware loop.

If you want to do this for an image, implementing say a 3×3 sharpening filter, then the idea is much the same. The first three cycles are just like above, then the next three cycles we load in three different kernel coefficients, and the input (i_n) values comes from one line up, then the next three cycles another three kernel coefficients, and the input (i_n) values comes from two lines line up. After these nine cycles of multiply-adds, we send the results to later stages, and move by N (so maybe 16?) along the image and repeat.

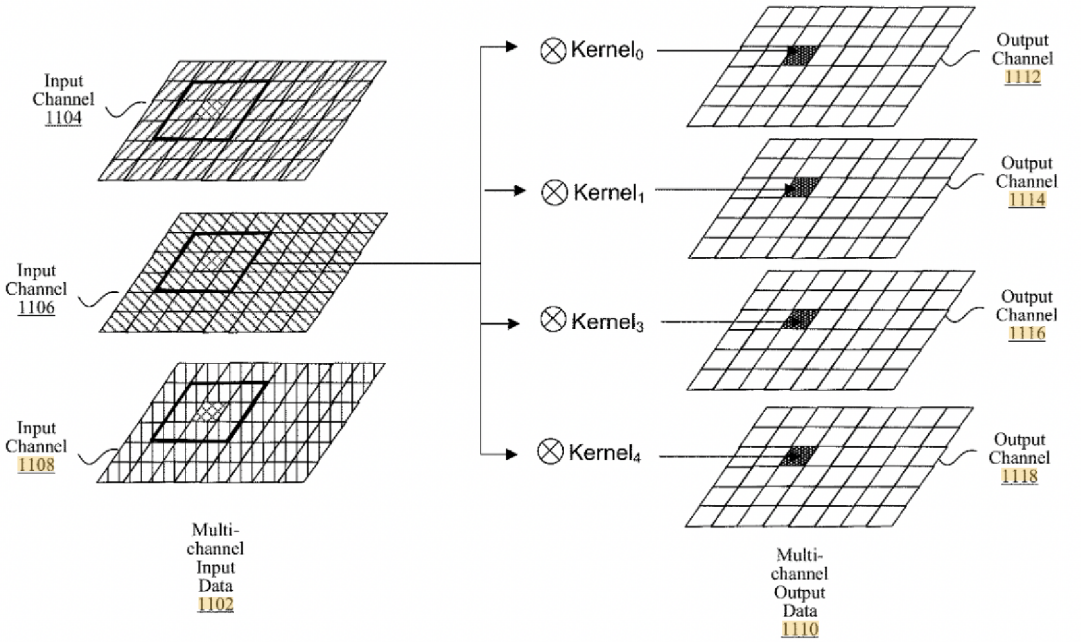
Given these primitives (most importantly the smart input buffer that can walk over the input data in the required pattern) we can do other things as well. Suppose, for example, that we want to implement a median filter. Now we need to walk over the image in the same way, but once we have gathered a 3×3 block of input data, we want to find the median. This requires some storage for each FE and a unit that can perform max/min tests. That's more or less what the Compressor (additional functions) and Multi-Cycle Acc do.

One final thing you need to remember is that the data flowing into the Vision block (and thus the Convolution Engine) is raw sensor data, so it's in Bayer format, interleaved. So one row of image looks like GR GR GR, then the next row looks like BG BG BG. The smart buffer has enough smarts to be able to do things like skip channels and walk along the array by 2 rather than 1.

So you can imagine, for example, a very simple pipeline that does something like
- first pass through feed to each FE the successive GR BG values of a Bayer Quad, multiply each by the appropriate color weightings, and generate a luma value. So input is a $2X \times 2Y$ sequence of Bayer values, and output is an $X \times Y$ luma array.

Alternatively the image sent in could be in separate channels RGB format. Conceptually we can define a $1 \times 1 \times 3$"convolution" that reads a single value of each of three RGB channels and multiplies the value by the appropriate weights to generate a luma value. This would be executed by defining the source as a rank-3 tensor (three planes of images) and the rasterizer feeding data into the multiply-adders would successively load a line of R data, then a line of G data, then a line of B data, allowing, after three cycles, the generation of a line of luma data.
- feed back the result (connect 428 to 422) to run the luma array through a high pass filter to accentuate edges
- feed back the result again through a median filter (now we don't do convolution, but we use the same sort of machinery to send the luma values from a 3×3 image patch to each FE which sends them on to Compressor which, after collecting all 9 values finds the median and outputs that value.
And so it continues.

More generally what we can do is things like



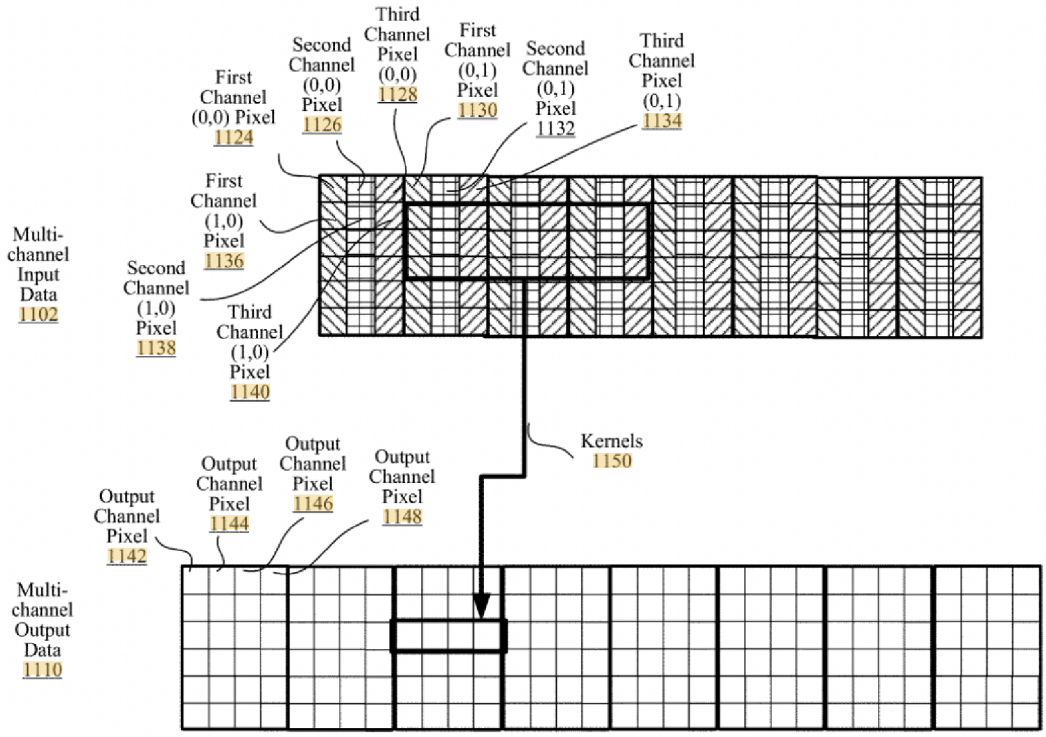
Here the idea is we have three input channels and we generate four output channels (based on four separate kernels). Each kernel is 3×3, so that the output value is generated over nine cycles by multiplying together 3×3 input values against kernel values.

If we have a few registers of local storage per multiply-adder (and we do, about 8 such registers at least as of the first ANE) we can implement this as something like
- load the required input values from the first channel
- broadcast the kernel value k_0 for the first kernel, multiply it by the first input, and store the result to

temporary storage 1 associated with the FE
- broadcast a different value k_0 for the second kernel, do the multiplication, store the result to temporary storage 2 associated with the FE
- continue for the third and fourth kernels, using two more temporary storage locations
- then load in a patch from the next input channel and repeat again, this time using the kernel coefficients appropriate to the (first input value from the second channel), accumulating against the store value in temporary storage 1.
and so on, and so on, in this case through (3×3 kernel coefficients ×3 input channels × 4 output channels)=108 cycles to handle the four output results (but in those 108 cycles we generated 256 results for each of the four output channels).

The point is that if we have even just a small amount of per-FE local storage
- we can “interleave” different convolutions in such a way as to repeatedly reuse a single Work Unit of input data to generate multiple partial products all being accumulated to separate partial sums.
- we can perform “convolutions” that are not just within a single channel, but in some way mix channels together
- as much as possible we can avoid the (more expensive) cost of repeatedly moving signal data to the FE and instead just keep updating the kernel coefficients which are broadcast to all the FE’s

The same sort of thing can be done, just with slightly different addressing details, if we, for example, interleave the three channels, and use them to generate four interleaved output channels. This is used when the system is taking data directly from the camera, and presumably it could be used if one wanted to interleave generic channel data, but most neural nets seem to prefer using separate planes of channel data.



This is still a fairly hardwired engine, though able to perform convolutions and thus to perform basic vision like detecting faces. In terms of interesting specific hardware details:
- there is some support for sparse kernels. In other words if some elements of a kernel are sparse, the various sequencers (one walking over the list of kernel values and broadcasting them to the FE's, the other walking over the input data) both know to skip this entry, avoiding the costs of data movement and multiplication.
- calculation are all fixed point, essentially as 8.8 math. Some calculation can be done just in integer space as one pass, some are performed as two passes, and the resultant two halves of the calculation are merged together by the Post Processor 528.

(2017) double convolution engine (two convolution "cores")

This scheme gets an update with (2017) https://patents.google.com/patent/US10176551B2 *Configurable convolution engine for interleaved channel data*. It's basically the same hardware, only replicated so that it can run faster (or more precisely do more work on larger images).
The new diagram looks like

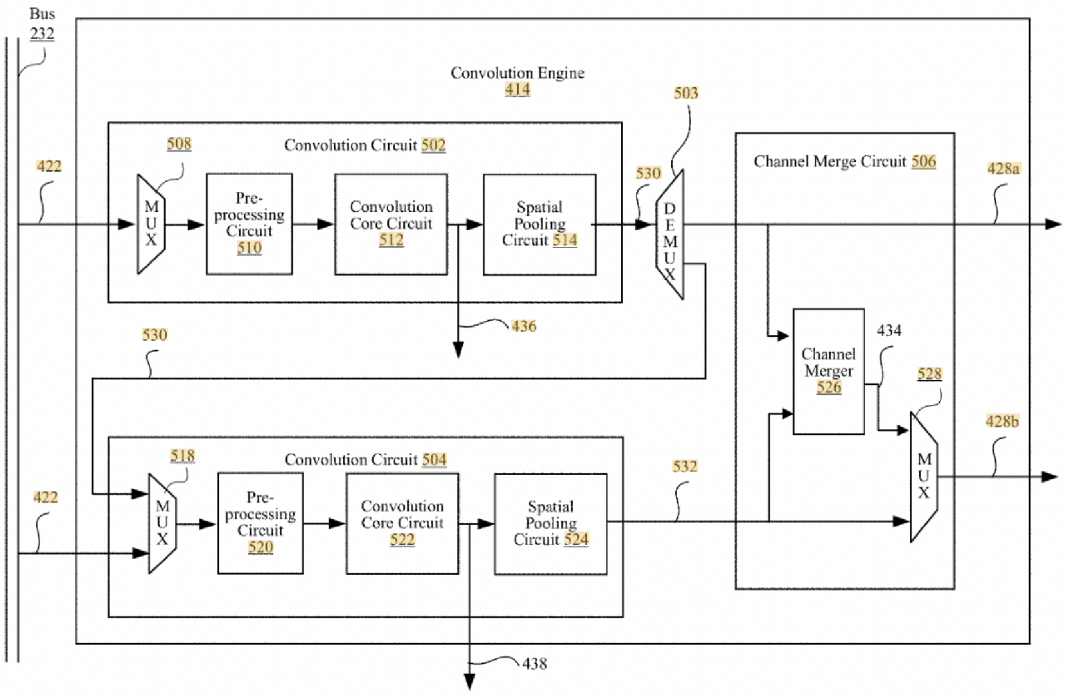
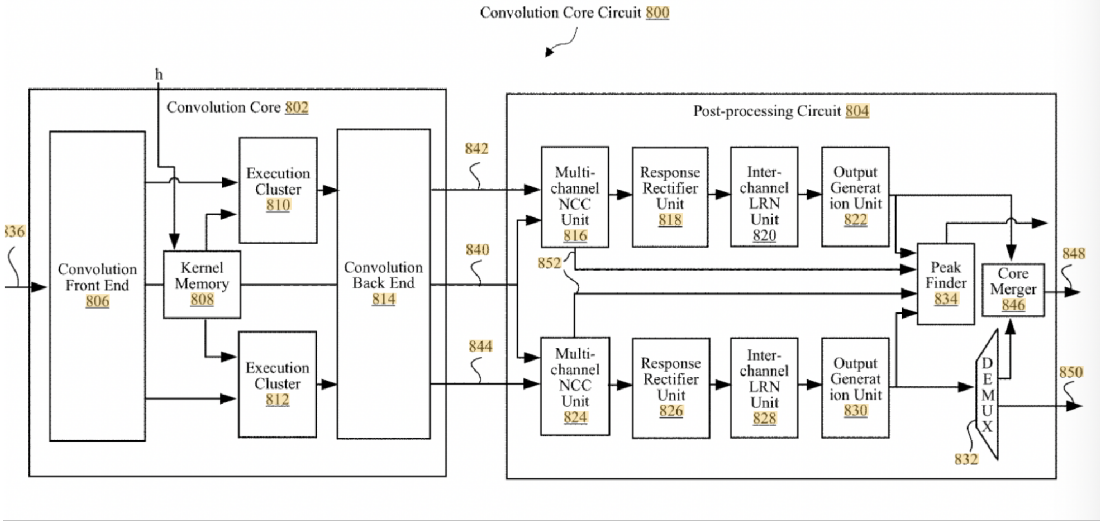

Bus
232
Convolution Engine
414
503
508
Convolution Circuit 502
422
M
U
X
Pre-
processing
Circuit
510
Convolution
Core Circuit
512
Spatial
Pooling
Circuit 514
530
D
E
M
U
X
Channel Merge Circuit 506
428a
436
530
Channel
Merger
526
434
528
428b
518
Convolution Circuit 504
422
M
U
X
Pre-
processing
Circuit
520
Convolution
Core Circuit
522
Spatial
Pooling
Circuit 524
532
M
U
X
438

FIG. 5

and


Convolution Core Circuit 800
h
Convolution Core 802
Post-processing Circuit 804
836
Convolution
Front End
806
Kernel
Memory
808
Execution
Cluster
810
Execution
Cluster
812
Convolution
Back End
814
842
840
844
Multi-
channel
NCC
Unit
816
Response
Rectifier
Unit
818
Inter-
channel
LRN
Unit
820
Output
Generat
ion Unit
822
852
Peak
Finder
834
Core
Merger
846
848
Multi-
channel
NCC
Unit
824
Response
Rectifier
Unit
826
Inter-
channel
LRN
Unit
828
Output
Generat
ion Unit
830
D
E
M
U
X
832
850

and

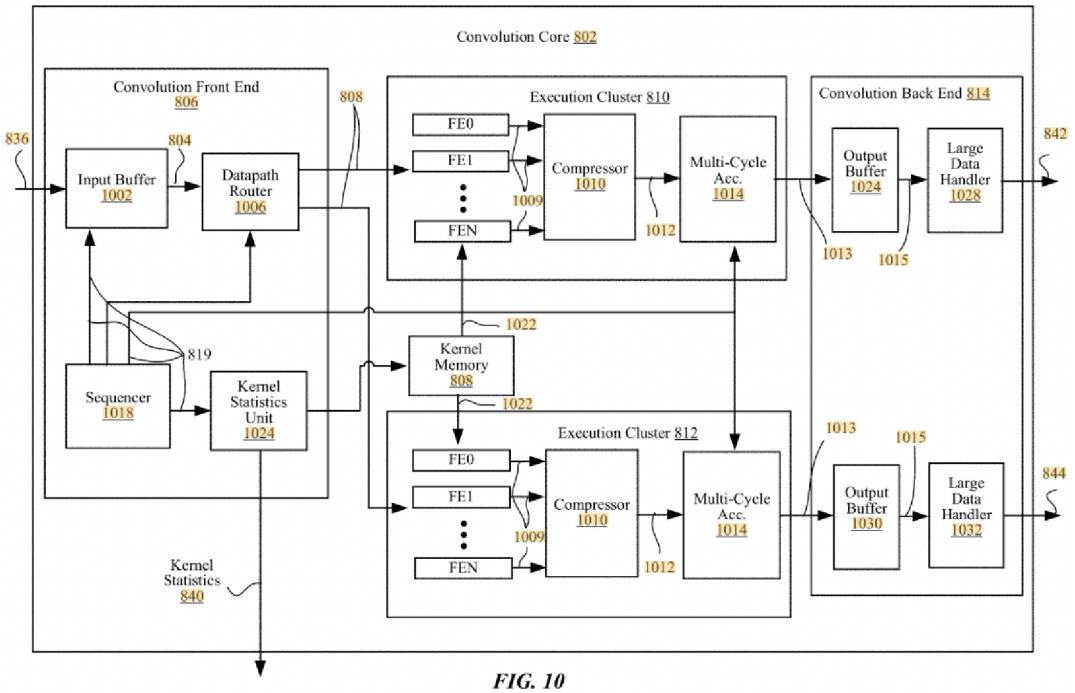


FIG. 10

So the points we immediately see are
- in the first diagram we now have two Convolution blocks. These can either operate in parallel (one handles top of the image, other handles the bottom) or sequentially, with one performing an initial transformation which is fed to the second for the next transformation.
- each “Convolution Core” has been augmented with specialized hardware. Spatial pooling (basically shrinking the image in various ways, possibly with the average of values over same small patch, possibly by selecting the max or min or median of values over that small patch) is now dedicated hardware. Pooling could be performed by appropriately using the earlier Convolution Engine hardware, but at more expense in power; offloading it to dedicated hardware saves energy and allows the area-expensive Convolution multipliers to be doing their thing for more cycles.

Likewise we add a dedicated ReLU lookup along the path in Post-processing Circuit 804.
- The “Convolution Core” now contains two “Execution Clusters”. So along with various specialized hardware we have what looks like 2×2 times as many multiply-adds available to us overall.

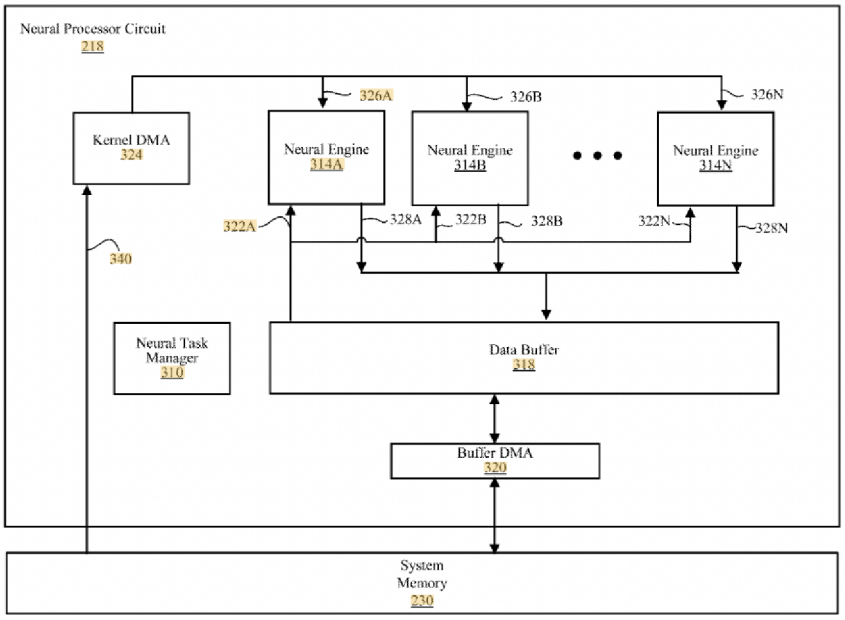
This is all still background; we’re still in the world of a fixed Vision pipeline based on dedicated hardware elements (eg Convolution and Spatial Pooling) and controlled more by “configuration” than by anything that looks like a program.
Everything is analogous to GPUs before programmable shaders were added – we’re trying to make the system more flexible by adding more fixed function units, and by making each fixed function unit slightly configurable, but it’s obvious that we need to rethink the model to allow for “programmability”.

(2018) The ANE Arrives!

background (overall design of the ANE 1.0)

After all this (substantial) throat clearing, we finally reach the genuine first version of the ANE with (2018) https://patents.google.com/patent/US20190340491A1 *Scalable neural network processing engine*.
It appears that Vision is now broken out to ANE, with clients that now extend beyond just the camera.

The basic ANE looks like so, a collection of “Neural Engines”



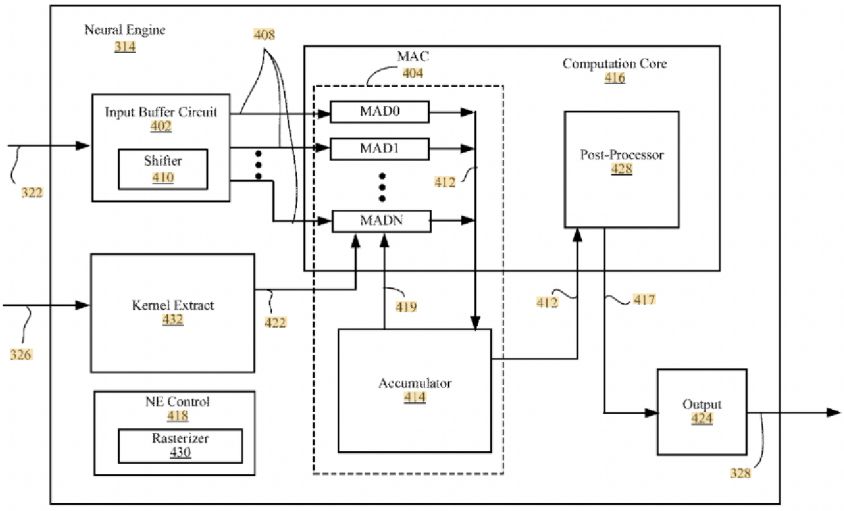
which each look some what similar to the earlier Convolution Engine. As far as we know, there were 8 of these cores in the A12 and A13 versions, 16 cores in the A14/M1 versions. If we compare this with the GPU, some similarities present themselves.
The Neural Task Manager is probably a companion core like the GPU companion core.
The Data Buffer plays something of the role of the GPU L2, as a buffer for data flowing in and out, and as a way for cores to communicate. However it is a *manually* controlled pool of SRAM, not a cache – the neural compiler manually commands when data is stored in the buffer or removed, and is responsible for ensuring that there is no overflow or address collision of two separate buffers. In this respect it is like the Scratchpad of the GPU.
Secondly, like the L2 of a GPU or the SLC, but even more so, the Data Buffer exists primarily for buffering and communication between different data sources and sinks. There’s not much of an expectation of data persisting in the Data Buffer for valuable reuse many many cycles after it was first placed in the

Data Buffer. So while we will see the Data Buffer picks up many smarts over the years, these are smarts associated with optimally routing data between producers and consumers and handling flow control between the two; but unlike a CPU L2, smarts associated with predicting the most useful data many cycles from now are not part of the design.

You'll notice that we have two DMA engines, and that's another difference. The ANE does not use "simple" load-store commands and so is not a "pull" architecture, where cores request data and then the core (or at least a thread on the core) sleeps till the data arrives. Rather it is a push architecture – the "program" interleaves DMA commands which describe the transfer of data from system memory into the ANE, and when that transfer is complete the next step of the program, some sort of "compute" command is issued to get the cores computing on the newly arrived data. Analogous to what we have seen with the GPU, as much "simple" work as possible is offloaded from the cores to dedicated hardware; in this case that means that some degree of data rearrangement and reformatting can be performed by the buffer DMA.

In some sense neural networks "transform" a stream of unknown "signal" data using a collection of known "kernels". Signal might be a stream of audio, an image, or a sentence. "Kernel" might be simple filters to extract edges, second stage filters to detect textures, or complex fully-connected stages where the feature vector generated by earlier stages is multiplied against a large matrix (the result of all that training we hear about) to determine some output – maybe speech to text, maybe a list of items detected in the image, maybe a predicted next word. The "weights" that we hear about (as in the size of ChatGPT, or as in some company has released their model weights) are numbers that live on the Kernel side of things.

Looking at each core we see something rather similar to the the earlier Convolution Engines, somewhat rationalized (IMHO) but still with the confusing names – which will continue...
We'll give the big picture first, then explain each element in a lot more
detail.



So two streams of data flow in, signal stream 322 from the ANE-wide Data Buffer 318, and Kernels. Output signal 328 can be looped around for a second degree (or more) of processing.
The basic execution of a layer (let's assume a convolution) is signal comes in (different values targeting each Multiply-Add unit), kernel values come in (same value broadcast to every multiply-add unit), the two are multiplied and stored in the Accumulator (a few elements of local storage for each MAD unit), then we repeat. After some number of cycles of this (eg 9 cycles for a 3×3 convolution) the convolution has been calculated for N elements (as many as we have Multiply-Adders) and these N output values are sent to Post-Processor which performs simple transformations on them, for example a ReLU lookup.
As a final round of processing before the data in relinquished, the Output block 424 may take the raw stream of output, 417, and resequence or reformat it into an order and format required by the next use of this data.

The neural compiler takes a set of layers as described earlier and essentially
- fuses together layers as much as possible. Usually linear layers (eg convolution or matrix multiplication) are followed by a delinearization function like ReLU, so those are trivially fused together. If the layer is followed by a subsequent convolution type layer, then the output from 328 can probably be looped around to 322 for the next convolution type layer. If the layer is followed by a pooling layer (which changes the size of the layer) this may not be possible.
- sets up various degrees of looping that are required to perform the (possibly fused) layer operation(s). So something like (for every channel of column of every row do ...) The innermost elements of these loops are handled by hardware in the ANE, but the outer loops will be handled within the program generated by the compiler.
- calculates for each layer the DMA parameters to move the signal into the ANE (move this many bytes from address A to Data Buffer Address B performing these data transforms along the way)
- likewise for the Kernel and the DMA parameters to move it into the ANE

Once a layer starts, to a substantial extent it runs "autonomously". The Rasterizer element 410 is essentially a hardware loop that loops over rows, columns, and channels as appropriate; the Shifter element 410 is the same sort of thing applied to ensuring that each cycle he appropriate data elements are moved out of the local storage (Input Buffer Circuit) for this core and into the Multiply-Adders. The Shifter is somewhat more complex because, as we have seen, it may have to incorporate a stride (for interleaved channels), it may have to move along the *x*-direction for a few data elements then shift backwards a few elements and move on to the next *y*-column, and so on (if this is confusing just look at the pattern for how data is read for a 3 × 3 convolution, especially with channel-interleaved data).
Essentially each core has hardware loops that operate on rank-3 arrays; task that require higher rank arrays perform the sequencing and slicing at a higher level, as an explicit loop within the ANE program carried by I guess Neural Task Manager 310. (It's possible that this very first version only handled rank-2 arrays with its shifter/rasterizer, and rank-3 arrays arrived a year later.)

Kernels (remember this is the known data as opposed to the unknown signal) are stored compressed, and so are decompressed by Kernel Extract 432.

This scheme allows data to flow into a core, be processed by one layer of a neural network, then flow out as Output 328, to be redirected into Input 322 for the next layer of the neural network. Note, however, that there is no direct flow from one *neural core* to another. All flow of that sort needs to proceed through Data Buffer 318 (analogous to communication via L2 between cores in a GPU). Better communication/co-ordination between cores will be one theme as the ANE evolves. Also it should be clear that the design *really* wants to move all overhead (looping and addressing, data sequencing and reformatting, lightweight lookup functions, etc) into small blocks of dedicated logic that kick in at various stages along the pipeline.

some numbers associated with the ANE

In terms of numbers
- there appear to be 256 of these multiply-add units in a core. This more or less seems to match the performance claimed by Apple.
$8 \times 256 \times 2 = 4096$, and if we assume the ANE runs at something like 1.2GHz this gives us ~5TOps/sec.

- as usual I am taking a multiply-add to be two “operations”. If Apple wanted a marketing number, they could easily pad this by throwing in extra operations performed by the post-processor, or by counting load/store and data rearrangement as operations. But they seem to be limiting themselves to multiply-adds.

- the Convolution Engine was essentially a fixed point design, optimized for 8-bit integer data, and with some hardware to split 8.8×8 (either 16 bit convolution components or 16 bit input data) into two separate planes of computation which would then be merged together.
The replacement ANE is a mixed precision device. The multipliers can handle either INT8×INT8 or FP16×FP16 multiply, but, unexpectedly, they can also handle a mixed INT8×FP16 (or the reverse) multiply! Later we’ll see how this is handled.
The resultant products are then accumulated as 32-bit wide (either INT32 or FP32) as temporary storage in one of eight (per MAD) storage locations in the Accumulator. When a convolution (or other sequential arithmetic, like a matrix-vector multiply) is complete, the values are transferred from the Accumulator to the Post-Processor where, among other things, they are rounded or normalized from a 32 bit format down to some chosen intermediate format . This accumulating a large number of products shorter precision elements into a larger precision accumulator is, of course, basically the same logic as nVidia’s use of TF32 in the Tensor Cores.

The 2018 set of patents we’re currently discussing, and multiple later patents, all continue to the Post-Processor as being able to merge planes of 8 bit results, in language that looks like it was cut-and-pasted from the vision engine patents. But I think this was just ignorant paralegals not realizing that

needed to be changed. There seems to reason for the current design to have retained the 8.8 fixed point support.

- The Input Buffer holding data to be fed into the multiply-adders can hold about 256B. This can be interpreted as anything from say an array of 128 elements each 2B in size to a 16×16 array of one byte data. So in terms of shape we can have shapes like say (for 1B data) 16×16, 32×8, 64×4, 128×2, to 256×1, likewise for 2B data. In terms of the optimal shape of the work unit, the compiler will decide that based on the shape of the convolution.
The Shifter will do the right thing in terms of extracting data appropriately and delivering it to the array of 256 multiply-adders.

The Buffer that I usually refer to as the "Smart L2" has been reported to be 4MB in size on the M1. Of course this is the sort of thing that could easily change in size across chips, even within the same family. For example the Apple Watch SoCs (which used to have 2 ANE cores and now have 4 as of the S9) probably have a smaller buffer.

Some more details are available at
https://github.com/tinygrad/tinygrad/tree/d0e752003da3fc023fa85094d7f5b65b47dd5091/extra/accel/ane

some benchmarks associated with the ANE

Looking at all this, and the number of data items available to be moved into the MAD units, suggests that *probably* there are 128 "computation units" each of which has two sets of INT8 math hardware and one set of FP16 math hardware. That seems obvious enough, but... Let's just check that hypothesis.....

It's difficult to get any data on the ANE but let's list what we appear to know.
https://apple.fandom.com/wiki/Neural_Engine lists the successive evolution. The Peak Ops appears to refer to multiply-adds but does not give precision. Ignore the value listed for A17 Pro – it appears to be a marketing mistake, and there's no good reason to believe the A17 is much different from the M3. Note that just counting these Ops, like the CPU or GPU, ignores significant other changes to the ANE, as we will describe.

Next let's look at ANE performance. The only reasonable source of data, imperfect though it may be, is GeekBench ML.
Here are three comparisons, each supposedly using the ANE (to the extent that it can be used, which is, for example, clearly not the case for the FP32 versions)., showing M3, M3 Pro, and M3 Max each compared to A17 Pro.
https://browser.geekbench.com/ml/v0/inference/compare/360358?baseline=360536

https://browser.geekbench.com/ml/v0/inference/compare/360618?baseline=360536
https://browser.geekbench.com/ml/v0/inference/compare/360661?baseline=360536

These numbers need to considered carefully and in context – the web interface is confusing and it's very easy, even for people trying hard, to refer to numbers that limit the benchmark to either CPU or GPU rather than using ANE. The benchmark also runs each subtest at three precisions and we can be certain that the FP32 precision definitely runs on either AMX or GPU. (Comparing M vs Pro vs Max suggests that it differs depending on the subtest, and probably some layers run on GPU, some on AMX).

The FP16 and INT8 results *probably* mostly run on the ANE, though things become less clear for the final (two times three) text-based subtests. When you look at how they scale, the primary factor seems to be SLC cache size, so that they run very well on Pro and Max (large enough SLC to hold the working set), and run only slightly better on M3 vs A17 (SLC not large enough to hold working set?)

Overall, looking at as many results as possible over as many SoCs as possible, a rough estimate since the A11 (averaged over many tasks, any particular task may be rather different!) has been that **for an A-class chip**, if the CPU's ML performance is x, then the GPU's ML performance is 2 x, and the ANE's ML performance is 3 x. But this is an average with very wide extremes, from ANE being the same speed to ANE being 4x or more faster, depending on details.
And of course (certainly for now, anyway)
- if your ML task needs to run on FP32, it cannot run on ANE.
- everyone (A to Max) gets the same ANE but Max gets many more CPUs and GPU cores, so as you go higher up, there is less performance win to the ANE.

So it's all very messy, but *one thing that is constant* across most of the tests and most of the chip designs is that *the F16 and INT8 results are very similar*. This suggests that either
- there are in fact the same number of FP16 and INT8 multiply-adders...
    In fact this is the case, there are 256 multiply-adders! We'll soon see the circuit design that allows for this and makes it sensible.
But there's still something weird going on here – we said that there's only 256B of "active" data available, so only 128 FP16 values can be shifted into the multiply-adders. Stay tuned for how this is resolved...

The size of the Input Buffer is stated in the patent as 256B a later patent shows that it is in fact 512B. As we will see in time, the "signal" eg a large image, is split into a hierarchy of smaller and smaller elements till we get to the minimal element that is processed by an ANE over one or a few cycles. At each stage of this hierarchy, elements have to overlap each other to some extent because filters and convolutions spread over the sides of the central element being processed.
So while the Input Buffer holds what is 256B of "real" data, there are available up to 256B to hold padding around the edges elements that will not "generate" an output element, but will go into the

computation of the values for “main” data elements.
Later we will see how this laid out and it will make a little more sense.

Of course all these numbers could change, though the only change that seems to have happened so far is the doubling from 8 to 16 cores.

So you can see performance as being something like
- every few cycles we bring in 256B (+padding) of data
- every cycle we extract (by horizontal and vertical shifts) some large subset of that data and send it to our 256 arithmetic units along with a broadcast kernel value
- a convolution will over a few cycles accumulate the output result
- which we then send to be post-processed
So for the simplest sorts of operations (eg just rescaling data or shifting data, so we just add or multiply the signal) we can pump through 256 values per cycle per core; for more serious operations like convolution we’ll generate something like values of data every 9 cycles or so (assuming say a 3×3 convolution).

One final point is that understanding just what the ANE can do (something we will be explaining in much more detail below) sheds light on decisions the ML compiler makes. People have wondered why some random net they pulled off Hugging Face is only compiled for the GPU or even only the CPU. The answer is that ANE (and to a much lesser extent the GPU) perform the most common layer types, not anything and everything.

So, for example, suppose someone experimenting with neural networks adds a function lookup layer that takes the logarithm of the input signal:

- the ANE has function lookups for the most common neural functions, but log is not one of these. It *may* make sense in terms of performance and energy to run part of the network on the ANE, then shift it to the CPU (or AMX) for the log layer, then back to the ANE. Or it may not; maybe it now only makes sense to run it on the CPU?

- if the log function is kinda useful, but the precise value of log is not really important, only a rough approximation to log, then what Apple might do in their use of this style of neural network is define a *pseudo-log* function which they add to the ANE.
Now we have a situation where Apple can get the benefit of log-like layers. *And* so can developers, *but* they have to be willing to modify their layer to use *pseudo-log* rather than log.

So it’s worth knowing what the ANE does well (which is presumably what the compiler will steer to the ANE), and being willing to modify your layers as much as possible to take advantage of this...
In this respect we are somewhat back to the early days of computing, and then GPUs. There will be people, the vast majority, who don’t really understand either their hardware or their software, and who will be getting 10% of the possible value from the hardware, and there will be the John Carmacks who *do* understand what they are doing (including understanding why any layer is the way it is, and the extent to which it can be modified, it doesn’t have to be exactly the way it appears on Hugging

Face) and they will be getting 100% of the possible value from the hardware.

If there is one thing you should take away from all this history and overall design, it is that ANE 1.0 was designed for *vision*, and is optimized to handle vision tasks (improving photos, recognizing elements in photos, detecting faces, that sort of thing). Not language! Of course language is the hot new thing, so much of what we will want to do over the next few years is improve things for language...

More Hardware 1.0 Details

Tasks (neural network), Task Queues, Task Descriptors (fused layers of a neural network)

OK, on to a deeper dive of these hardware elements.
Neural Task Manager maintains Task Queues. A task is essentially an entire neural network; the Task Descriptors sitting in each Queue are essentially the (generally two or three fused) layers of a network. It seems like the transition from one Task Descriptor to the next is the point where preemption can occur.

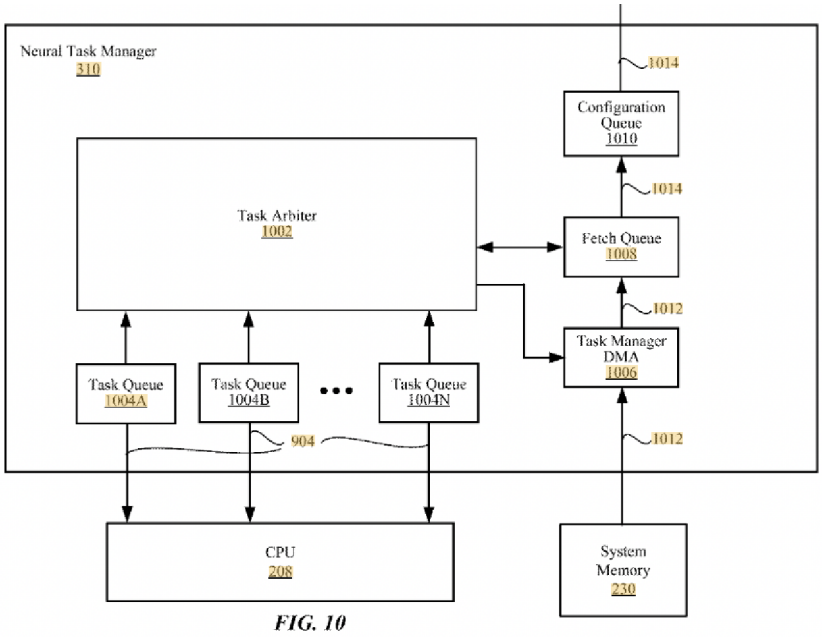


FIG. 10

So we already see a few things we recognize from our GPU evolution:
- right from the start there's an ability to submit multiple tasks, not one task that takes over the ANE until it is done
- there is a concept of priority so that an incoming higher priority Task can grab the ANE until it is satisfied.

Not covered in anything I have seen are some other questions:
- do separate processes submit tasks, or are all tasks effectively submitted by some sort of Neural

demon?
- is everything done in physical address space? That would certainly be cheaper in energy terms, and may be feasible given that ANE code is so constrained in what it can do (eg no concept of generic data lookup, so no way to peak to the data of some other neural net). However who does the mapping from virtual (as in, ultimately, the user signal submitted to the network) to physical and where does it happen?
- can separate Task Descriptors (ie different layers), either from the same network (ie two independent layers; we will see examples of this) or from two separate Tasks run simultaneously. Right now I don't think so but it's unclear. The patent make a point of saying that the kernel data supplied to the different Neural Cores can be different, which is part of what's required; but it's unclear if the "code" executed by each Core can be different. (One reason you might execute the same code, but using different kernels, on each core is when performing large matrix-vector multiplications, something we will get to.)

Next look at the stuff on the right hand side.
What sits in a Task Queue is a Task List which is a list of references to Task Descriptors. As resources are freed up within the ANE, Task Arbiter decides on the next Task Descriptor to execute. For now this seems to be a fairly static scheme (again things will probably change as the ANE becomes more important) but in this first design each Task Queue has a fixed priority, so we'll start executing one Queue and continue with it until completion, unless a higher priority Task comes in.

Once we decide on the next element in the Task List of interest, we send the Task Descriptor Reference to Task Manager DMA which loads the Task Descriptor (ie loads the equivalent of the code/-configuration to execute for this layer).
This Task Descriptor is loaded into "Fetch Queue" (better thought of as Fetch Buffer).

Once the Task Descriptor is fully loaded, it is committed to the Configuration Queue.

Once committed, it will execute, the stage between Fetch Queue and Configuration Queue is the last point at which preemption can occur. From the Configuration Queue, as resources become available the layer will be executed, essentially by programming control registers in the various parts of the ANE.

First as buffer space and kernel space free up, signal and kernel DMA's can be launched; then as the execution of the previous layer ends, the configuration of the various hardware elements occurs (setting up the rasterizer and shifter, setting the math units for FP16 vs INT8 operation, etc) then the next layer starts executing.
We can see an expanded version of this here: Conceptually each of the blocks in 1010 is a value to be transferred into some target register in one of the target blocks.

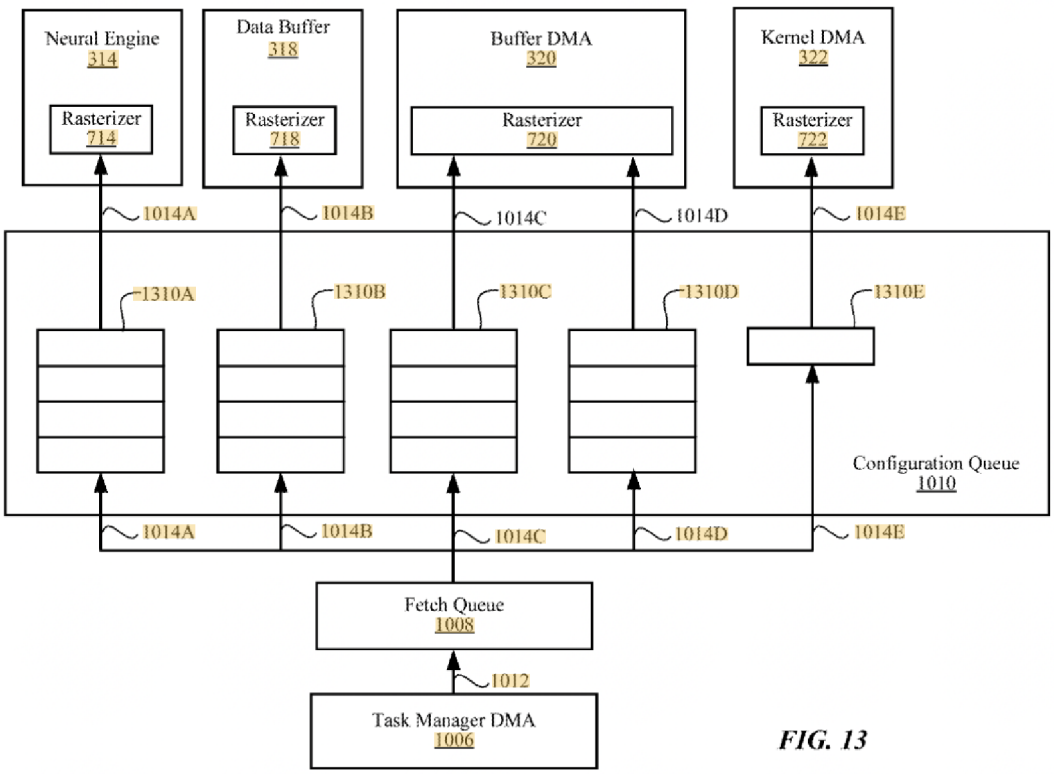


FIG. 13

Note also something that's easy to forget – some of the DMAs being programmed are not just to load data into the Data Buffer, but also to write out previously calculated data. That's why we have two queues into Buffer DMA. Obviously as far as possible you try to balance things somewhat along the lines of "load in data for new kernel, begin new kernel, while new kernel is executing write out data for previous kernel, then begin loading data for the successor to new kernel". Separately the queue in Data Buffer programs the rasterizer that moves data from the Buffer (after it has been placed there by either Buffer DMA or an earlier pass) to the Work Units passed into each of the eight cores.

The patent does not say so but an obvious assumption is that, like the GPU, all the registers in hardware like the rasterizers are duplicated, so that these can be programmed in advance then, with a single bit flip, switched to start executing the new layer.

The patent strongly suggests that (again as of version one) the Descriptors locked into Configuration Queue execute in-order, but an obvious future possibility might be, obviously subject to dependencies, the ability to rearrange the launch order of these items based on when their DMA's become available.

There's an expectation that this design can be scaled (probably along the same lines as how the GPU is scaled on an Ultra, with multiple ANE's working together). We've seen that the first round of this for the GPU was a disappointment, and some work was required to optimize scheduling across the two physical GPUs to minimize the synchronization and data transfer required by the two halves.
There may be some of this also for the ANE but, as we will see, it's an easier problem, because the design and data flow are much more constrained.

a more detailed look at a vision network

Backing up a little, let's look at a Task List.

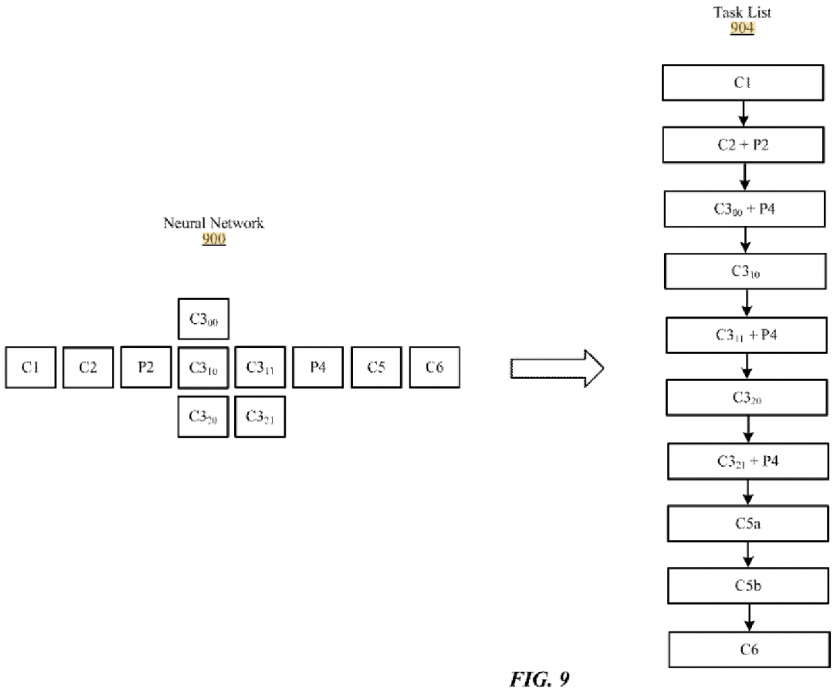


FIG. 9

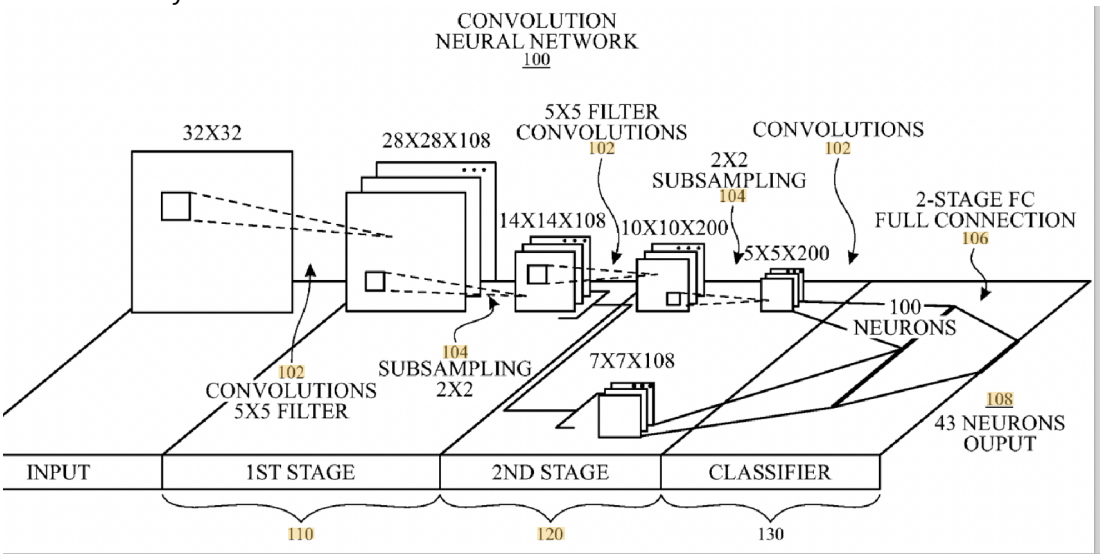
We see that this network has a few Convolution steps (C_n) along with a few Pooling steps (P_n). (Between each layer there will probably also be a delinearization step like ReLU, but as we have seen that's usually fused with the preceding layer, and the function lookup just happens trivially as the results flow through the Post-Processing block after the main work is done.)
We'll talk about Pooling later.

Note also that this particular network (like most real networks these days) after P2 runs the results through three independent convolution layers (and then feeds back to the main network via P4 pooling, but let's ignore those details).
So, *in theory*, we could try to do things like allow $C3_{00}$, $C3_{10}$, and $C3_{20}$to run in parallel, subject to data availability or whatever. In reality we don't even attempt this with this version 1.0 ANE, but stay tuned for how the design evolves...
So we linearize the list of layers as show to generate a Task List. Note that (and again this will change in later designs) we can fuse a convolution layer followed by a pooling layer into one unit, so that element (basically something like C, R, P, R) forms the basic unit of work that we apply to one block of signal that is loaded into a Neural Core.

Data flowing out of the Data Buffer (ie the L2 equivalent) can either be broadcast to all or multiple Neural Cores, or to just one.
Thus we can either apply a common kernel to variable signal (eg same convolution applied to a large image split over Cores) or a variable kernel to common signal (eg multiplying a large matrix, split over

all the Cores to a single feature vector broadcast to all the Cores).

We'll still get to matrix multiply, but before that, look at this image showing a simple vision CNN, for example something around the time of LeNet or AlexNet. This shows in more detail what's meant by some of the layers:



As you should know, an appropriate convolution kernel can detect edges. A slightly different such kernel will detect edges at a slightly different angle. Thus when we talk about the first layer of a net performing a 5×5 convolution on the source image we actually mean a set of many such convolutions. In the diagram above we start with an image (let's say just one plane of luminance, so it's a 32×32 image. We apply 108 distinct convolutions to this image, each time generating a different output, so that we end up with a rank-3 array of data, $28 \times 28 \times 108$. (The 32×32 image is shrunk to 28×28 because, rather than handling the edges specially, we just start the filter inset by 2 from each edge.)

We then perform pooling (for example max pooling, or average pooling) to reduce each 2×2 block to a single value, giving us now a stack of 108 planes, each 14×14 in size. What does this mean? The stack of numbers 108 high at each of the 14×14 points is a feature vector describing the lines present in the 10×10 block of pixels (in original 32×32 space) that surrounded this particular element of the 14×14 elements.

The second stage more or less repeats the process. We split the data in two, and one copy of the data we pool again to give us an even smaller representation of the lines present in each patch of the original source. With the second copy we perform more convolutions, which are now essentially extracting how rapidly one set of lines changes to another set of lines over this 5×5 (or, in original pixels, 10×10) patch. Since we move from 108 to 200 planes, various options for the exact details are possible – maybe most of the planes go through the same two kernels; maybe every plane goes through a different set of kernels, a few through one kernel most through two kernels? These are details that are learned by designing the network.

Regardless, after pooling we now have two feature sets of interest, one, 128 elements long,

describing the lines in each patch of the image, the other, 200 elements long, describing the textures at each patch of the image. In principle we could even map each element of these 128 or 200 into an atlas to show something of what type of line or texture that particular element is describing, but that's just to satisfy our curiosity, it not relevant to the operation of the network.

It should also be pointed out (and is important to functioning of the network) that between each stage we perform some sort of delinearization which could be thought of (depending on the exact delinearization chosen) as, for example deciding that a given line is either present or not present, no wishy-washy "sorta present to an extent of .47", and likewise for textures. Seeing these delinearization steps as binary decisions (present or not) or perhaps slightly more subtle (not present, $\frac{1}{3}$ present, $\frac{2}{3}$ present, fully present) explains to some extent why we can get away (in the right places, **not** everywhere!) with very low resolutions in various elements of the neural net (either as weights, or as quantization of the signal as it passes between various layers).

Finally given this total collection of features we have created is sent through two fully-connected layers (essentially multiply a large matrix against the vector representing the features, delinearize the result to get a set of intermediate features [which maybe have some interpretation, but I don't think anyone has worked this out], multiply this new feature vector (now features in some sort of "word space") by a second large matrix, and the final result is more or less a vector of numbers. Each element in the vector corresponds to a word (eg "cat", "frog", "house") and the number is something like "degree to which the image looks like the particular word").

Once again go read a specialized text if you want to know the details; the point of this exercise is to show that when you hear that the ANE is optimized to perform, say 5×5 convolutions, you tend to think of a single convolution, and sure, that's a lot of arithmetic, but moving the kernel around is no big deal. But I hope the above makes clear that, no, you're actually performing 108 convolutions, or 200 convolutions, each with different kernels, and later you're performing very large matrix-vector multiplies.

how the hardware is optimized for a standard vision network

All this explains why there are two mechanisms in place to try to shrink kernels, to reduce the data transfer required (as always, it's data transfer that generally limits your performance and costs the most energy).

The first method is that kernel coefficients can be looked up in a table. So we can, for example, quantize our coefficients for any particular layer into say one of 64 values (which could be, for example, 64 FP16 values), then define each kernel element using a 6-bit index; we provide the table as part of the data for the layer, we pack the set of 6 bit values into a dense bitstream, then the Kernel Extract part of the Neural Engine unpacks the bitstream and looks up the indices in the table. This is what people are talking about when you see examples on Twitter of "I encoded LLAMA using 6b and managed to fit it and run it in realtime on my M1 Max"; the very large classification matrices of fully connected layers of these language models flow into the ANE as kernels, and are "decompressed" by the

Kernel Extract block.

The second way to shrink kernels is that many kernels (whether for convolution or for fully connected layers, or for various other uses) have a large number of zero coefficients. Each kernel is described by a bitmap which marks the elements in the kernel as zero or not. Zero elements are skipped over by Kernel Extract.

It's an obvious question as to how aggressively we use the fact that we know some of these kernel elements are zero.

The least aggressive policy would be simply to save bandwidth in transporting kernels from memory to the ANE.

Slightly more aggressive, but still fairly easy to implement, would be to suppress the Input Buffer sending data to multiply-adders whenever a kernel element is zero. This would still take a cycle of time (during which the Shifter would internally increment as appropriate) but we would not pay the costs of moving signal data or kernel data to the multiply-adders, or the cost of their execution. My guess is that this is where Apple is with at least the first ANE.

Even better would be the ability to "fuse" skipping a zero with a subsequent execution (of a non-zero element). This would require the shifter to be able to run twice in a single cycle, but would also allow us to, for example, run about twice as fast if the kernel is 50% sparse (a not unreasonable situation).

Best of all would be if the kernel extractor could inform the shifter of a run of, say 7 zeros, and the shifter could perform the equivalent of looping seven times and then a final eighth time to deliver the data to the multiply-adders. This best of all version would obviously be a lot more complex, and whether it's worth doing would depend on the extent to which we see long runs of zeros in the kernels (and large matrices) that we care about.

Between these two extremes, depending on how fast we can run the shifter, there may be scope for an intermediate solution, like we can skip up to three successive zeros if we can run the shifter four times in one cycle.

As a technical point, the patent is somewhat confusing on one issue. The words of the patent fairly explicitly say that there is a single Kernel Extraction unit for the entire ANE, which presumably stores decompressed kernels in some local storage, or maybe generates them on the fly. On the pother hand the diagrams show a separate Kernel Extract unit for each Core... I've no strong opinion on the matter, but given the sorts of use cases where you'd want to sent different kernels to different Cores, having a separate per-core Kernel Extract unit (and on-the-fly kernel decompression) seems to me the more sensible choice.

Another takeaway from the description above of a very simple neural network is that it shows some examples of how you might want to mix and match the different functionality available. For example consider how to implement the first stage of 108 convolutions.
It's probably equal in performance, while better for energy to do something like
- broadcast the same *signal data* to eight cores
- have each core execute *eight separate successive kernels* on its data, accumulating the results in eight separate stores in the Accumulator.

This will allow us to generate 64 of the 108 required convolutions with minimal movement of the signal (one time load into the buffer, one time broadcast to the Cores); then we can move the 64 channels of output data through post processing and into Buffer storage; and do the same thing again with the remaining 44 convolutions (so 5 kernels sent to 4 of the Cores, 6 sent to the other 4).

Note also that even if we start put nets with "natural" image sizes, eg 32×32, the combination of dilation (losing elements at the edges because of how the convolution begins inside the image so as not to worry about padding) and then pooling means we are soon dealing with weird image sizes like 28×28, 14×14, and 7×7. We will eventually see the consequences of this, for now we'll just accept it.

(It's possible that at some point, once the frantic speed of current AI/ML slows down and we have time to go back and actually *optimize* all this stuff, some of this will go away. With time to consider various options, it may turn out that using appropriate edge padding [zero padding? replicating the edge elements outward? mirror padding?] works as well as, or better than, dilation; and will give us a better fit to computer hardware...

But for now in many cases we're stuck with people wanting to use the classical nets, not optimize them for particular hardware, and so of course Apple has to work adequately well in such situations.)

I've referred to the arithmetic units as multiply-adders because that's how they're shown in the diagram, but apparently with ANE 1.0 these units can also perform a few other arithmetic tasks, including specifically max and min. This means that they can perform a task like pooling. The Shifter simply has to walk over the set of say 2×2 source values feeding them to the arithmetic units one per cycle. The arithmetic unit performs a max against the value stored in the accumulator and stores the new max there, and after four cycles you have a max-pool value for this 2×2 patch. Note that in this sort of use you reuse data from previous cycles and stored in the Accumulator, but you don't need any input from the Kernel storage.

However the way pooling is handled in the ANE 1.0 seems clumsy and suboptimal. It's using expensive hardware (an FP multiply-adder unit) to do something very simple. We'll soon see a much better solution.

The Accumulator is better thought of as a few registers of storage (say eight or so elements) associated with each arithmetic unit, and able to be used in various ways. We already discussed a few times how this can be used to interleave a few separate kernels acting on the same input data.

Another feature of the Accumulator is that the storage is either double-ported or physically duplicated; the point is that the patent stresses that the Accumulator can be during every cycle be in active use in multiple ways;
- one way is being read (perhaps in the first half of a cycle) as input into the Multiply-Adder; and then (perhaps in the second half of a cycle) being written to by the Multiply-Adder
- a second way is being read from as output into the Post Processor block.

Finally we're left with the Post-Processor block. This is a somewhat random collection of circuits that can do a few different things and (I think) can do most of them in a single pass, just by passing the data through the first step then the next then the next.
The sorts of operations that can be performed include
- some easy non-linear functions (eg Abs or ReLU)

- some tree reduction functions. Suppose, for example, that you want to know the mean of this 16 × 16 patch of data that you have just processed. You need to add together 256 values. The MAD units do not have cross connections that allow for this sort of operation. But in principle it's not that difficult; you have 128 adders and then (depending of your performance goals) these can either feed into 64 adders which feed into 32 etc; so you can generate one reduction (pipelined) per cycle; or the adders can loop around so the outputs of the 128 adders feed into the top 64 adders, and so on; and you take seven cycles to get your reduction.
These sorts of reductions are necessary if you want to know patch means, patch variance, and cross correlations (so things like [deviation of R from the R-mean] times [deviation of G from the G-mean] summed over all pixels in the patch). This is called NCC (Normalized Cross Correlation) and is useful for image processing and camera usage.
Another version of this might use min or max tree reduction, which allows us to generate the largest (or smallest) value within a patch or from a set of channels at a single pixel. This is part of what's called LRN (Local Response Normalization) and is a common layer in many neural networks

All of these reduction operations can, one way or another, also be achieved by the MAD units at the cost of a lot of input shifting, over many cycles, but it's cheaper in energy to do this in some hardware that's wired up for the task, and doing this allows for more aggressive layer fusion, so that the successive work units from the same layer can pipeline through the ANE, having some part of the task (convolution and then max pooling) performed in the MAD units, then moved to the Post Processor for ReLU then maybe LRN or maybe NCC.

Once again, in ANE 2.0 we'll see a much better solution to this task.

- a non-obvious job of the Post Processor, not discussed until a much later patent, is rounding. The MAD units appear, in floating point mode, to accumulate to FP32. That is, the input data (signal and kernel) are two FP16 numbers which are multiplied, and the result is treated as an FP32 number and added/stored as such. This allows some degree of intermediate accuracy if we expect many terms to cancel each other out to leading order. The same sort of thing more or less holds in INT8; again larger intermediates are accumulated in the Accumulator allowing for the values to temporarily get larger than INT16 (8×8 int multiplication) but then cancel out as some negative values come in.

At some point, however, these high accuracy values need to be rounded to the format used by the rest of the network, and this rounding, whether of FP16 or INT8, is performed very early on (probably the first step) by the Post Processor.

It's never stated explicitly, but my guess is that at this stage the INT32 values are also rounded back down to INT8, probably by some user-set shift amount, like shifting the INT32 value by 9 or 10 bits. Various properties, including rounding mode, and other details of what the Post-Processor should do, are stored as part of a Task Descriptor (basically the "machine code" for each specific layer of the neural net).

how signal is split into slices,tiles, and work units fed into the ANE

Next let's consider how the signal is divided up. We'll start with the simpler diagram, and describe things in terms of images (2D planes), though most of these ideas extend in an obvious way to more dimensions.

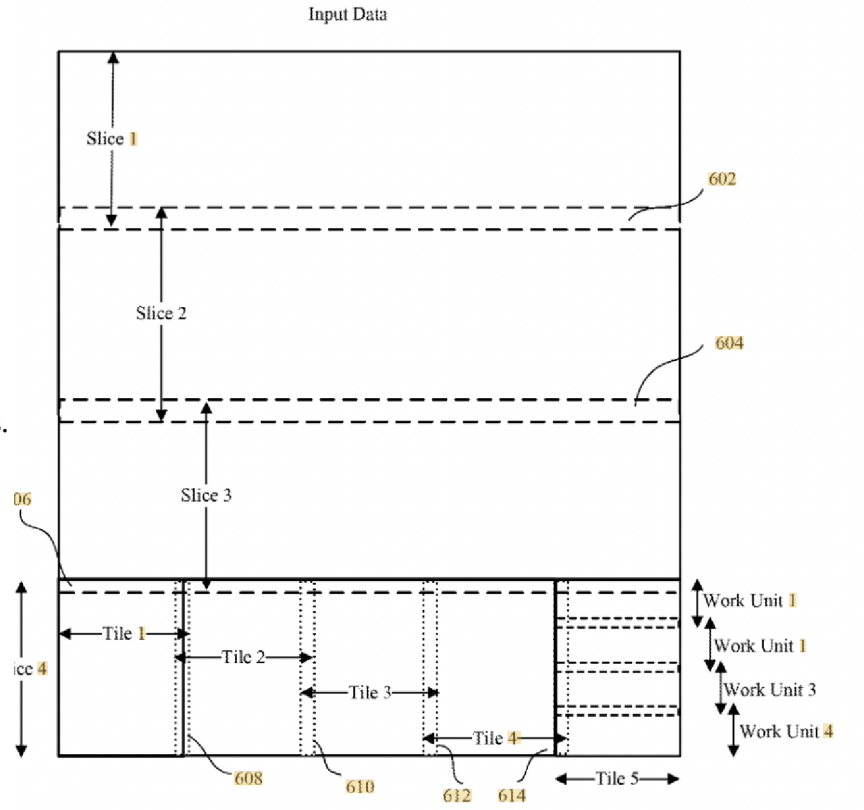


FIG. 6

An input image is segmented into Slices which are segmented into Tiles.
- Tiles are the unit that is transferred by DMA into the Data Buffer.
- Slices are envisioned in the being the unit cached in some higher level cache that communicates with multiple separate ANE units. Obviously this is the role played by the SLC, and it would be natural to use all the cache control machinery of the GPU (DSIDs, marking subsets of cache data as purgable, all that sort of thing).

It's unclear if the ANE uses the full set of options available as aggressively as the GPU, but the patent describing the ANE compiler talks about a Memory Optimization module that, among other things, generates hints to control how the SLC handles ANE data transfers.

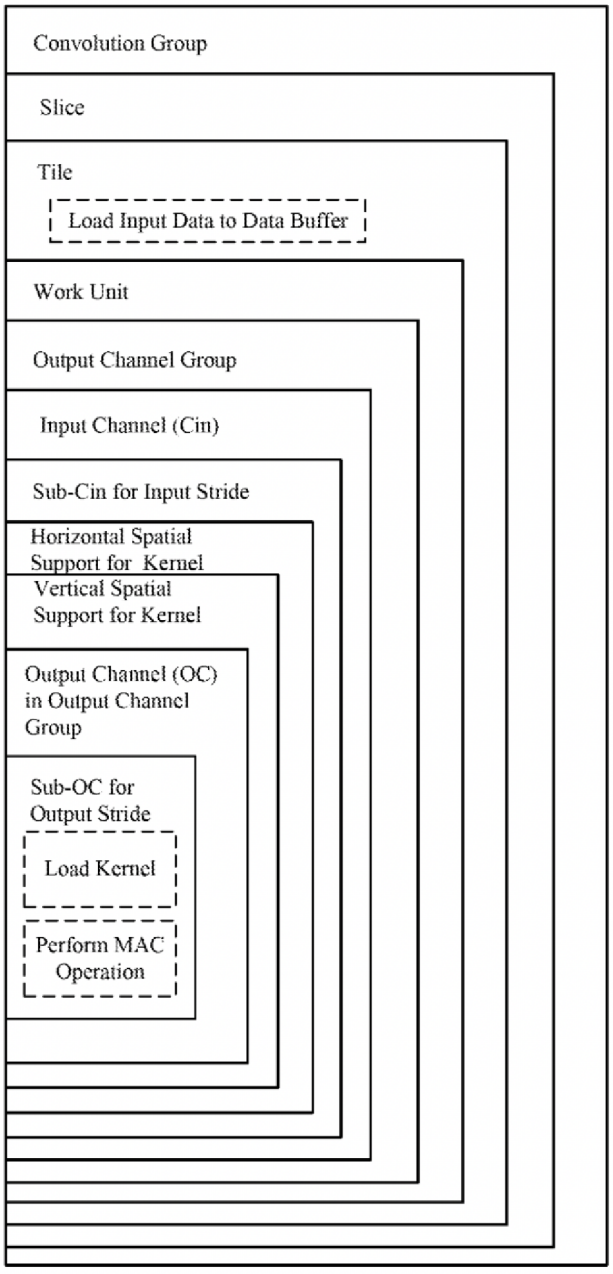
In the context of an Ultra machine, separate slices will also be handled by separate ANEs in parallel, eg ANE1 handling Slice 1, and ANE2 handling Slice 2.
- Tiles are split into Work Units which are the nominally 256B units sent to an individual Neural Core.

You will notice that each element in the hierarchy has some degree of overfetch, to the extent

required by layer being executed; for example a 5×5 convolution will require two pixels of overfetch in each direction.

What about edges? For now (...) these are lost opportunities. If Tiles 1..4 are say 32 pixels wide, and Tile 5 is just 20 pixels wide, then smaller Work Units will be submitted for Tile 5 and while (I assume!) we won't lose energy (we won't transmit unneeded data, the relevant MAD units will not be activated), and everything will work correctly, we will however lose the computation opportunities of these cycles.

This hierarchy of data units fits into this diagram which shows the hierarchy of loops:



This can make your head spin, but it's easy enough to understand when you think about it, and it's all about maximal data reuse.
We load some data into the MAD units and then (as much as possible) reuse that data multiple times for multiple kernels (ie the innermost loop is of successive kernels applied to the same signal data). This is because kernels (broadcast the same value to all the MAD units) are cheaper to move than signal data. This is also why we want to run up to 8 kernels interleaved with each other (calculate the partial product for one kernel, then change to the next kernel and calculate it's partial product, and so on) storing the intermediate partial sums in the Accumulator, rather than finishing a kernel and moving on to the next one.

There's also a slight degree of flexibility in the system. Some kernels or pooling require "skipping" data values, so the shifter handles this, and this same skipping means that interleaved channels can be handled, in which case interleaved channels would move lower down the hierarchy, to just above the Output Channel. However mostly it's easier to think of non-interleaved channels, and the system certainly supports that.

Most of the rest of the details can be ignored, the main point of interest is that these are the elements that are tracked by the hardware. Rasterizers in the DMA track slices and tiles coming into the Data Buffer, and Work Units being sent to the different Neural Cores; Rasterizers in the Kernel Extractor and related hardware track the current horizontal and vertical position for each active kernel; and the Shifter for the Input Data tracks horizontal and vertical position and whether any sort of horizontal or vertical stride (which is also used to handle interleaved channels) is relevant.
There's also some degree of hardware loop support within the hardware for looping over channels, so ultimately "rank-3 support" within the hardware itself. Higher ranks are handled by software, analogous to how you might handle a cube of data by sending one plane at a time to specialized 2D hardware.

handling matrix-vector and matrix-matrix multiplication

So this gives us the basic design. We now have a number of patents issued at the same time that amplify various aspects of the design or show how it can be used for unexpected things.
Let's start with the most important one. This hardware looks great for convolutions, and can also handle pooling, but what about multiplication of a large matrix with a vector, as required by fully-connected layers?
That's (2018) https://patents.google.com/patent/US11487846B2 *Performing multiply and accumulate operations in neural network processor*.
Suppose, for example, that we wish to multiply a matrix (signal) against a vector (kernel). To be definite, let's say the vector is 100 elements long, and the matrix is 128 by 100. So the output will be 128 numbers each of which is a 100 element dot product.
In the first cycle we load the first column (128 elements) into the multiply-adders, broadcast the first vector element, and multiply.

Next cycle we load the next column of 128 elements into the multiply-adders, broadcast the second vector element, and add each product to the previous product. Repeat 100 times and we have accumulated 128 dot products as required. This is easily and obviously scaled up (or down) as required.

This sounds great, but there is a problem in that the more usual case of interest is multiplying a feature vector (ie signal) against a large matrix (ie kernel). I cannot understand the explanation the patent gives for how to do this.
The basic problem is that the kernel machinery all seems to be based on broadcasting a single kernel value to the entire set of FMA units, and that doesn't work, it just doesn't have the bandwidth, for any reasonable scheme that wants to route the matrix (much more data than the vector) through the kernel path. Each cycle we need to generate (regardless of the details) 128 new products, which means somehow 128 new data elements on one side or the other of the multiplication. We can load in 128 new elements each cycle from the data side, but nothing we see elsewhere suggests we can load more than one new element each cycle from the kernel side.
So I don't know! Presumably there is something that is not explained in any of the patents, some sort of alternate mode that allows a separate, different, kernel value to be sent to each FMA unit rather than the usual convolution broadcast mode? If you can figure it out let me know.

I stress this point because, at the end of the day, while vision has its value, it's language that's the prize for AI/ML, and language requires the multiplication of a vector (representing signal) against large matrices (representing the "internal structure" and "semantic connections" of language). We need matrix-vector multiplication to be fast and low energy, and I don't see how the current design does it well enough.

But the gap between what we have and what we want does not seem *that* large. Right now we have tied together
- kernel DMA, kernel storage, kernel extraction, and kernel broadcast (limited bandwidth)
- signal DMA, signal shifting, and signal transfer (high bandwidth)

We can break these apart! Instead of defining a signal path and a kernel path, define a "matrix" path and a "vector" path, each of which can behave as either signal or kernel.
Store both kernel and signal in a single pool of SRAM, and along both paths provide an extraction unit (to handle appropriate ways of compressing kernel, whether it's convolution kernel or a large but sparse fully-connected-layer matrix). This should give us most of the value of what we want.

The one remaining issue is how best to exploit sparsity on the matrix side. Right now we can exploit sparsity on the vector side well, and that's probably good enough for now; later we can consider options for doing better on the matrix side when the matrix is known (ie forms kernel rather than signal). At the very least we could suppress the data motion and FMA for the known-zero lanes...

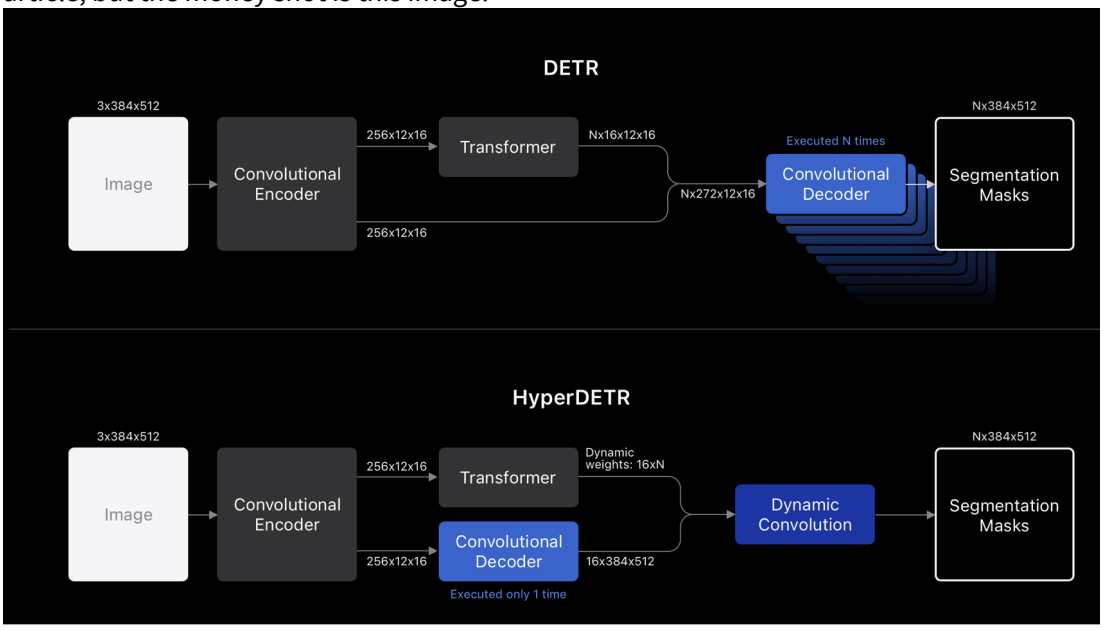
Once you vector matrix multiply working, then matrix-matrix multiply is a trivial extension, just multiplying one row, or column of one of the vectors at a time against the other matrix.

One possibility for solving the matrix-vector multiplication problem would be if we could dynamically treat signal as kernel, in other words if we could dynamically generate data which is written out to the "L2" Buffer storage, then re-read as kernel weights. It's not obvious that we could do this because (as

the next section describes) kernels are stored in a compressed form. Assuming we don't care about this compressed form (and so can't take advantage of sparsity) can we then read generated signal as weights?

Apparently so!

Look at (2021) https://machinelearning.apple.com/research/panoptic-segmentation *On-device Panoptic Segmentation for Camera Using Transformers*. There's lots of technical ML language in this article, but the money shot is this image:



The upper neural net shows the original image segmentation workflow; the lower net shows a restructured version optimized for ANE.

The image demonstrates two things.

First, that we try to run two parallel subnets independently, potentially simultaneously on different ANE hardware elements.

Second, and relevant to what we just discussed, the Transformer subnet is generating dynamic weights, (ie constructed as signal) which are then routed as weights to perform the second "Dynamic Convolution" element to generate the *N* segmentation masks.

The patent also describes how to perform tensor products (trivial) and element-wise operations (eg adding two matrices to each other). However we won't discuss this because the next version of the ANE makes the hack this uses unnecessary.

One final element that's missing from this discussion is the dot-product of two sparse (say 5% filled) vectors, which is an essential primitive for multiplying sparse-sparse matrices. (We have good support for a non-sparse signal dotted against a sparse kernel, but not for a sparse signal dotted against a sparse kernel.)

Apple has not (yet?) provided hardware for this, but I'll discuss the issue towards the end of this article.

kernel compression (mask out zeros, and quantized weights)

The patent (2018) https://patents.google.com/patent/US11120327B2 *Compression of kernel data for neural network operations* gives some details of the kernel compression scheme. We start with a bitmap that indicates zero vs non-zero coefficients, something like:

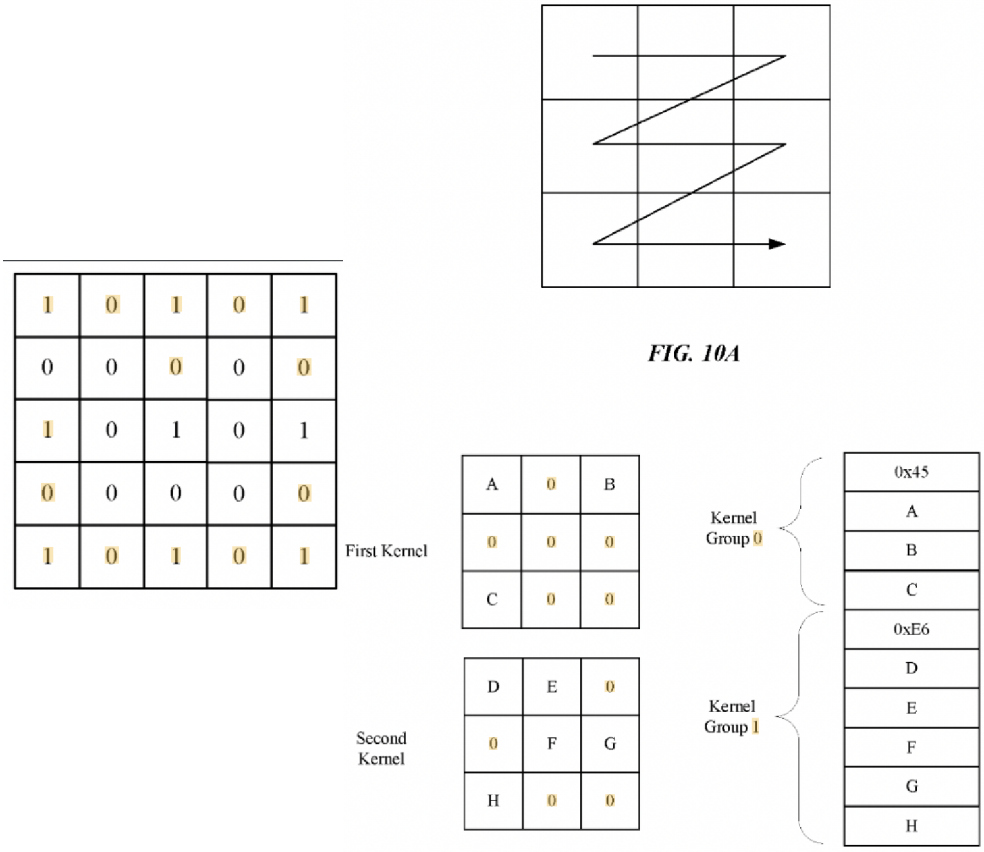
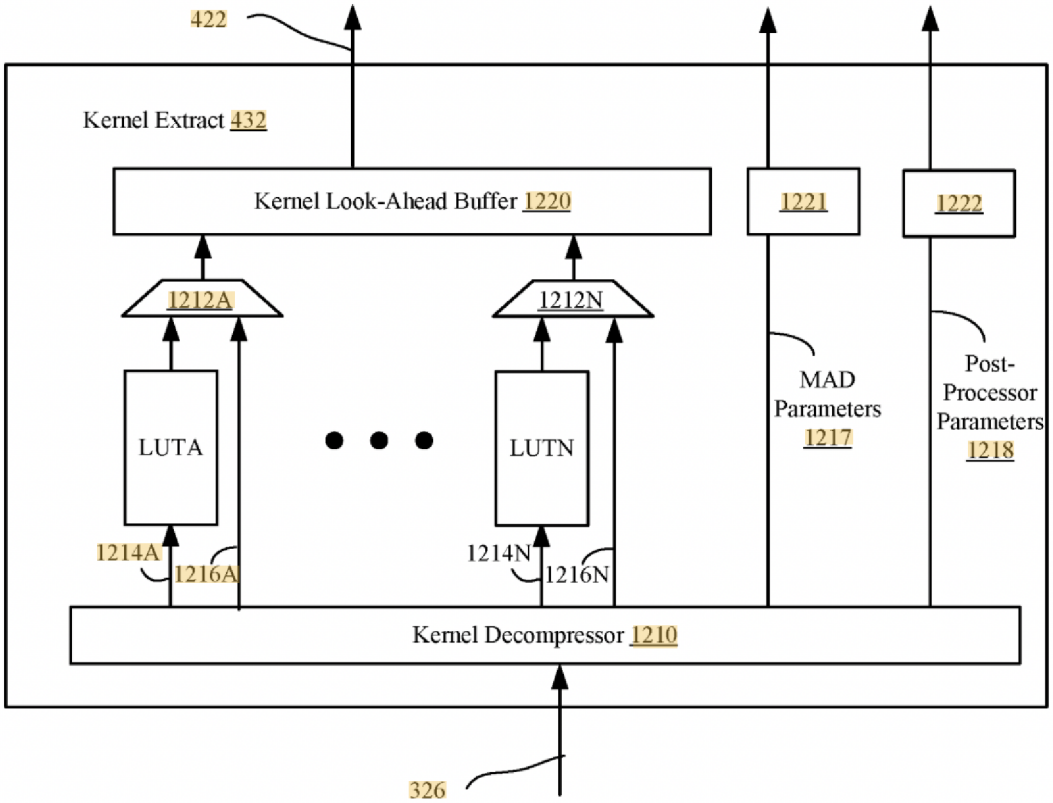


FIG. 10A

The bitmap is scanned in a defined order, and the bits grouped by 8 to form bytes that are interleaved with the actual data values as shown. For a kernel group all to be applied together (think of our example of 108 different kernels detecting different edges in a small window) the successive bitmasks (which might be say 9 or 25 bits in length) are strung together and divided between kernels in a way that means some bitmask may be associated with the previous or next kernel, but we don't waste any bits. So the 0x45 and 0xE6 above ultimately (once you lay the bits out in the correct order, and allow for the fact that any trailing bit mask bits of all zeros can be omitted) correspond to describing the non-zero bits of the two kernels shown. In principle the same idea extends to higher rank. The patent talks about a rank-4 kernel, but given that the entire inner core of the ANE is based on rank-3 tensors, my guess is the scheme is only implemented up to rank-3.

Of course, as already mentioned, the actual values A, B, C, ... can be indices into an FP16 lookup table. The ML compiler can cluster kernel values together to create optimal values for as many entries as you're willing to allow in your lookup table.

There's a little more to Kernel Extraction than you might expect:



Data flows in (326) from Kernel DMA and flows out to Multiply-Accumulators. The design appears to be that the ideal use case (with some variants possible) is
- the same workgroup (signal) gets broadcast from Data Buffer to every core of the ANE
- the same kernel gets DMA'd to every core of the ANE
- but each core extracts a different subset of of kernels to apply to its copy of the workgroup
- which is why we have separate copies of Kernel Extract in each Core

Kernel extract does the obvious things (parse the bit stream, look up bits in a lookup table, and insert zeros) along with some non-obvious things.

One of the non-obvious things is that there is some flexibility for different lookup tables for different kernels within the kernel group. (Presumably details like this are all laid in in some header at the beginning of the compressed kernel group.)

Another non-obvious thing is that MAD-parameters ultimately boils down to an initial non-zero value that can be seeded into the Accumulators before we start, to add a constant offset to the convolution being calculated. In ML language this is called *channel bias*.
Likewise Post-Processor parameters can perform some (I assume per-kernel) toggling of how we want the data to be treated (de-linearization function, rounding mode, and whatever) as it flows through Post-Processor.

Look-Ahead Buffer (as always with the names that make very little sense...) is the part that

interfaces with the rest of the core to handle zeros, so with the Input Shifter (to ensure that we increment the shifter loops but don't move data) and with the FMAs to ensure they don't compute.

The patent also clarifies that we do make some attempt to save power and execution cycles by not moving data or doing work in response to known-zero kernel values; but does not clarify how aggressive the scheme is in terms of the options I suggested.

Another weird issue is that nothing anywhere indicates some sort of buffer to hold Kernel DMA, however such storage does exist. It's even possible to reuse a kernel that has been loaded into that kernel storage, though I have only seen reference to this possibility, no details of how the "instruction stream" handles this re-referencing.

how a single arithmetic unit handles both FP16 and INT8

(2018) https://patents.google.com/patent/US11880757B2 *Neural network processor for handling differing datatypes* allows us to solve the riddle of how INT8 vs FP16 operation is balanced.
Here's what the FMA unit looks like when operating in INT8 mode:

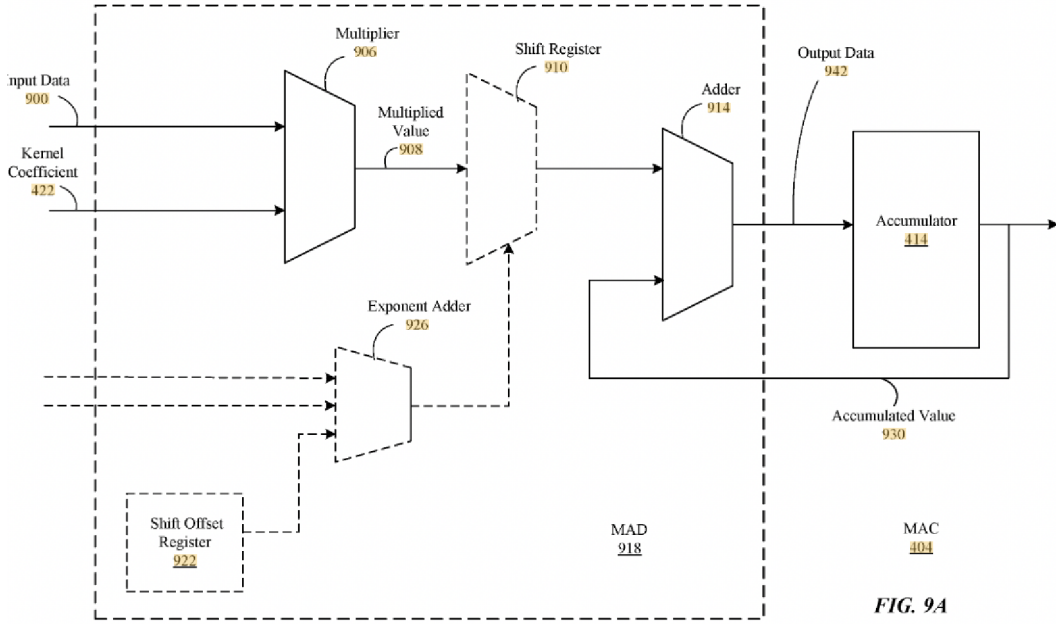


FIG. 9A

Simple enough – input data and kernel are 8 bits, they route through an 8×8 bit multiplier, all the dotted part are bypassed, and the 16b product is routed to the adder where it is added to a 32-bit accumulating sum. In other words the temporary sums that accumulate in the Accumulator are 32b integers.

Now let's look at FP16 mode:

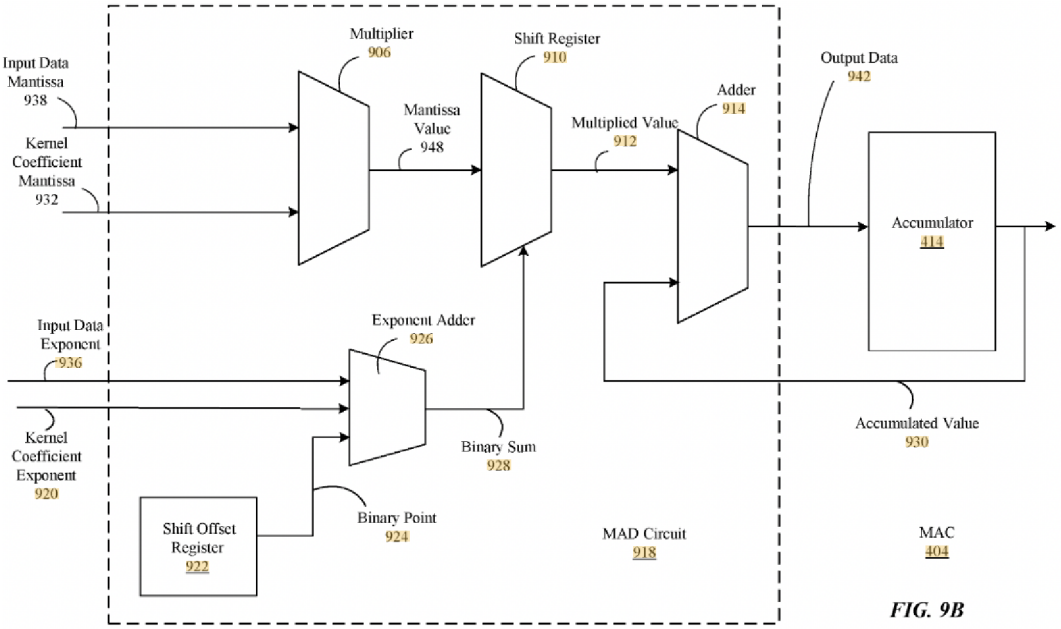
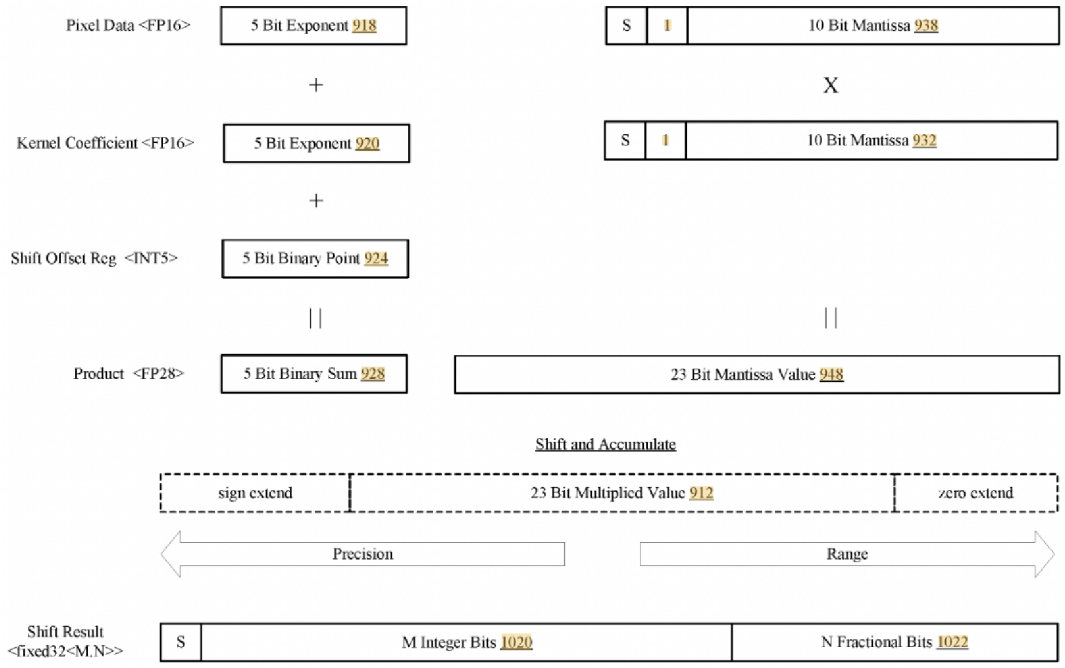


FIG. 9B

The exponents and mantissas of the signal and kernel inputs are split, and the usual steps of FP multiplication are handled.

As long as we have an 11×11 multiplier and keep track of the exponents properly we can share hardware.



We then accumulate the results, now as something that's basically FP32-like.

In fact we can do even better. The patent states that you can run the hardware in three modes, either as INT8/INT8 or mixed INT8/FP16 or FP16/FP16. A mixed INT8/FP16 mode (either side can be either

precision) is simply a question of faking the correct values in the exponent to make everything line up properly!

This is presumably what Apple means by the mixed ANE supporting "mixed" precision. The early (camera Vision) patents talked about 8.8 mode, and how that was handled, and there's still reference to that in a block of verbiage describing the Post-Processor in every patent, even the most recent 2023 ones. But I think we have to conclude it's obsolete; the early Apple Vision stuff ran an 8.8 fixed precision mode; but the new model is to use INT8, FP16, or, when it makes sense, both.

There are still two final elements you may have forgotten!
First, the patent clarifies that the Accumulator holds 256 × 8 × 32B storage locations, ie each multiply-accumulator has 8 storage slots. The words used (which don't have to correspond to how you actually utilize the hardware!) is that there are 8 *channels* of storage available, and the expectation, as I've already described, is that you will interleave eight separate convolutions on the same input work item before you move on to the next input work item. This works when, for example, you load in one image plane to generate 8 (of 108 final) edge detector kernels.
You can also split the storage into two halves, so that only 4 channel are available, but the other 4 channels can be used to move data on to some later part of the ANE (or to loop around and pass Post-Processed data through a subsequent convolution, which may be possible if there is no pooling/reshaping between the two layers, only some delinearization).

how 128 input FP16 values utilize 256 arithmetic units

The second point is what about the disparity between 128 input FP16 values and having 256 FMA units? Ahh, this is both very obvious, when you see it, and very clever!
Here's how we wire things up.
- we transfer each input value to two FMA's, eg FMA 0 and 1, then FMA 2 and, then FMA 4 and 5
- instead of just executing just one kernel, we execute two successive kernels, call them A and B from the workgroup
- so instead of broadcasting one kernel coefficient to all 256 FMAs, we broadcast the coefficient for kernel A to the even FMAs and the coefficient for kernel B to the odd FMAs
- so we have half as much input data, but we execute twice as many kernels and so the INT8 and FP16 rates are balanced! The only cost is enough hardware to extract two kernel coefficients per cycle rather than one, and some routing logic to broadcast either one value to all the FMAs to two values to the even vs the odd FMAs. Very nice!

support for alternative formats (BF16? FP8?)

You can see that there's nothing here that makes it at all difficult to add BF16 as a format, should that be desired. Apparently that has not yet happened (at least I can find no evidence for it or reference to it, even though BF16 was added to NEON, AMX, and the GPU with A15/M2.

It would certainly also be possible to just support FP8 and FP4 formats (presumably the ones that

seem to have finally been settled upon as of 2024, not the entire zoo of possibilities!). But it's much less obvious how you could generate value from these formats! The way this FMA is set up doesn't naturally lend itself to giving twice the FP8, or four times the FP4 throughput that nVidia provide with the way they have wired up their Tensor cores.
(Which doesn't mean it's impossible! But I suspect it's a lower priority than many other things still to be done to ramp up ANE performance. Remember that, already, compressed kernels mean Apple gets the memory bandwidth benefit of things like FP4, with somewhat more flexibility in the 16 values chosen rather than the 16 values defined by the FP4 format; so to the extent than ML is frequently limited by memory bandwidth, the most important problem is already solved.)

Finally apparently these FMA units can also handle basic comparison (ie generate whether input A is larger or smaller than input B). This is presumably done by the same sort of circuit that we saw for the GPU, a circuit that can handle both integer and FP comparisons [as opposed to the solution of subtracting the one value from the other, which does more work than strictly required, but can just reuse an adder].

processing multiple independent convolutions on the same input data

(2018) https://patents.google.com/patent/US11200490B2 *Processing group convolution in neural network processor* is not very interesting except that it states something of the obvious, that there are multiple ways we can split a problem across the ANE; and it confirms our understanding of what's possible.
Group convolutions is the situation I keep describing (as a nice reference example) of 108 edge detectors; so where one input is acted against by multiple independent convolutions. The ANE can handle this is multiple different ways, and of course normally the compiler will choose the optimal strategy.
So, for example, one strategy might be to process the entire image [split across multiple cores] as one convolution, before we move on to the next convolution. This splits input, but broadcasts coefficients. You might do this if you only have a few convolutions that you need to apply to the image.
An alternative strategy (the one I suspect is optimal for energy, and for performance unless something blocks it from being used) is to reuse the same work item in each core, but process a different convolution on each core. This broadcasts input, but splits the kernel.
(Then more aggressive versions of both of these would in fact use the multiple storage locations of the Accumulator to process multiple kernels interleaved, so in the first case, up to 8 kernels interleaved against a split image; in the second case up to 64 kernels (8 cores each running 8 different kernels) against the same segment of the image broadcast to all cores.)
And of course you could mix and match these (eg use four of the eight cores for half the image, four of the eight cores for the other half).

handling interleaved data (probably *only* relevant for camera input)

(2018) https://patents.google.com/patent/US20190340498A1 *Dynamically shaping and segmenting work units for processing in neural network processor* talks about the finicky details of how you handle

interleaved data. I suspect this is only interesting for the specific case of camera data flowing into the ANE in Bayer format, and anyone else will treat their data in the much simpler format of separate, stacked but not interleaved, planes. So only bother looking at this if you really care about Bayer format.

task switching (not yet very sophisticated)

(2018) https://patents.google.com/patent/US20240069957A1 *Systems and Methods for Task Switching in Neural Network Processor* discusses something of pre-emptive context switching, ie switching partway through the execution of a net, between layers, in response to a higher priority network being submitted. The main not-quite-obvious item of interest is that source data for the next layer may be DMA'd in from SLC/DRAM, or it may be stored in the Data Buffer (ie L2-equivalent) having been placed there by an earlier layer. We have to track this and handle it appropriately. There are some not very interesting protocols followed by the various pieces to try to optimize this, ensuring that, for example, if a switch is coming up, then the target address for data that is generated will be altered from an address targeting the Data Buffer to an address targeting system memory (ie SLC/DRAM), and likewise once the net begins running again, it will read that data from SLC/DRAM rather than the original source address of the Data Buffer.

some optimizations performed by the compiler

This is getting exhausting, I know, but we are almost done! The last patent
(2018) https://patents.google.com/patent/US11640316B2 *Compiling and scheduling transactions in neural network processor*
cover aspects of compiling for this sort of hardware. Nothing much interesting here because it's mostly known compiler techniques. Perhaps the most challenging element is tracking data usage because the Data Buffer is not a cache, so the compiler has to pack as much into the Data Buffer as possible while ensuring that data in use is never overwritten by, eg, either the output of Post Processor or a DMA operation.

A fair degree of optimization is performed, though it's optimization at a different level from say a C-compiler. The most important optimizations include
- merging layers into a single Task Descriptor, ie a single data pass through the ANE
- merging memory reshaping (eg tensor transposition or extracting slices from a tensor) into the DMA description, so that there's no real separate performance of this memory manipulation; the appropriate data, appropriately re-layed-out, is simply placed in the Data Buffer, ideally while the ANE cores are busy computing away on earlier data
- optimal memory reuse of data within the Data Buffer without having to transfer to SLC and back
- embedding of hints for the SLC in the instructions given to the DMA.

Along with this, if compiling is your thing, you may want to look at (2019) https://patents.google.com/-patent/US20200082274A1 *Compiling models for dedicated hardware*, which discusses the issues

involved in splitting a neural network to run partially on CPU, partially on GPU, and partially on ANE.

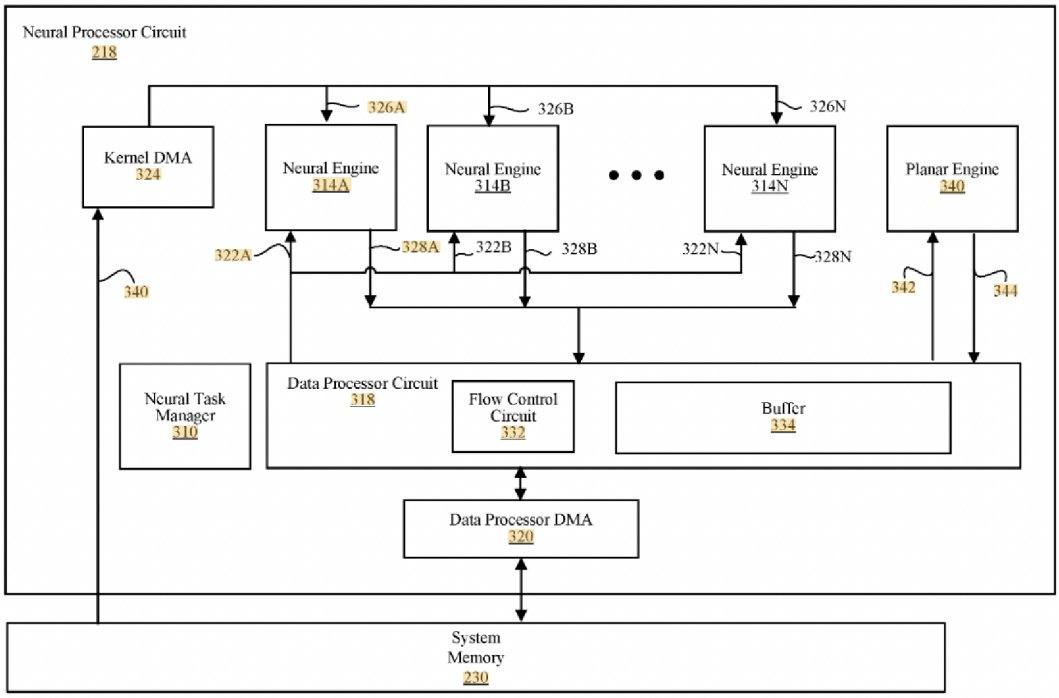
(2019) ANE 2.0. Addition of the Planar Engine and “Smart L2”

Congratulations! At this point you now know more about the ANE than practically everyone on earth outside Apple. What’s next?
We can improve the ANE in the obvious ways – boost the frequency a little, increase capacity of the Data Buffer, add more Cores. That’s uninteresting.
We can add more precision support (BF16 and some version of FP8 should be easy) and that might be valuable.
But it’s always the less obvious stuff that’s interesting (and that’s frequently where the real win is).

If we think about the GPU, then the obvious places to think about are
- sometimes we are forced to used our expensive arithmetic hardware for what are fairly trivial tasks like pooling. Can we offload these trivial tasks to some simple hardware?
- how often are hardware elements (whole Cores, or a subset of FMAs) going unused because the size of a problem does not match the full 8 Cores, or 256 FMAs per core? Can we do anything about that?
- are we being optimally aggressive in terms of pipelining problems and using every element (FMA, Post Processor, DMA engines) simultaneously; or are there circumstances that frequently cause one or the other of these element to wait?

These sorts of ideas that forms the basis of the next few years of work.

So let’s consider the 2019 update, starting with (2019) https://patents.google.com/patent/US20210103803A1 *Multi-Mode Planar Engine For Neural Processor*. The overview diagram looks like



If you compare this with 2018 there are two changes.
The first is the addition of the *Planar Engine*,
the second (mainly in support of this first change) is that the Data Buffer/L2 equivalent has been upgraded to a “smart buffer” in ways we will describe.

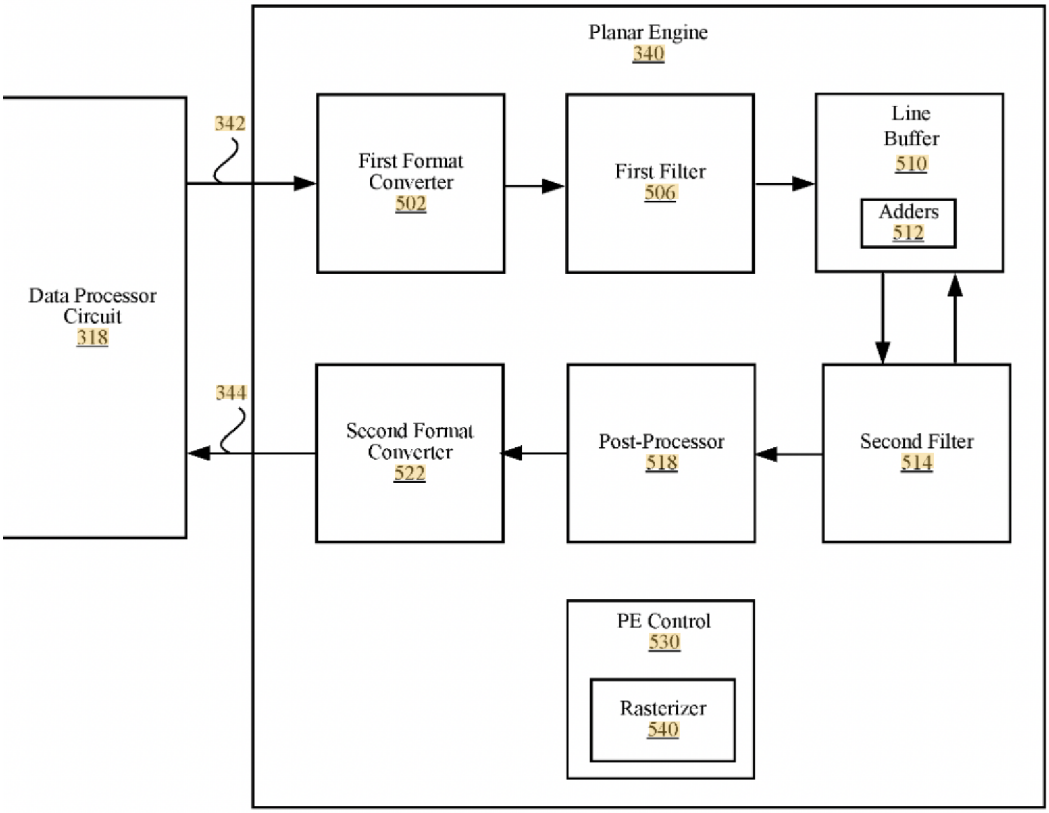
The Planar Engine

The Planar Engine is many things. It’s a way to offload work from the Neural Engine Cores to some additional hardware, but there are other ways to think of it.

One way is that the Neural Engines can natively handle up to three dimensional tensors (for example they can perform 3D convolutions); the Planar Engine is optimized for 2D work (though it can handle most of the equivalent 1D operations).

Another way to look at it is that the Cores handle complex work that is primarily limited by computation; the Planar Engine handles simple work that is primarily limited by memory bandwidth. (This raises an interesting issue – which memory bandwidth? The 50 GB/s or so of the A17, or the 400GB/s or so of the M3 Max? The fact that Apple hasn’t yet decided to scale the ANE as we move up the product line the way the P cores, the GPU, even AMX, are scaled, is an interesting and non-obvious business decision. I assume it’s something like Apple doesn’t want to make supporting this hardware any more difficult than it needs to be, not until some sort of critical mass of apps and developer understanding has been generated? Regardless, I assume the ANE is essentially scaled for the A17, so is pretty much always the optimal choice there, and for power, but may not be the optimal performance choice when you have a Max available.)

The Planar Engine looks like



So data flows in or out via the Data Processor Circuit (the new name for what used to be called the Data Buffer, and which I think of as analogous to an L2). The big picture is that it can go through some light data remapping on the way in and out (the “Format Converters”) some more substantial “Filtering” and some generic “Post-Processing”.
The diagram is somewhat misleading. There are two blocks that act as “Filters” which can be accessed in either order, and these blocks can store some temporary data in the “Line Buffer”.

It’s never pointed out explicitly, but a variety of patents suggest that both First Filter and Second Filter consist of something like a 16×16 block of arithmetic units, the most obvious difference between the two being that one of these arrays is wired to allow easy communication vertically, the other to allow easy communication horizontally. Later we will see a second significant difference between the two.

In the 2019 patents the implication is that these filters are something like 8×8 in size. Later patents suggest they are 16×16 in size. Of course the precise size does not matter for understanding the idea, and it could be that a later revision increased the size from 8×8 to 16×16 (perhaps at the point where the number of ANE cores was increased from 8 to 16?)

general behavior (including pooling mode)

Some of the tasks the Planar Engine can perform include
- Pooling of various sorts (eg max or mean pooling)
- Various reductions (eg collapse rows or columns of a plane to a single column or row)

- Normalizing a plane of data (gather statistics, and/or add a bias and rescale the data)
- Elementwise operations, and simple 1 or 2D filtering.

First Format Converter can perform simple functions like ReLu, multiplying by -1, or taking the absolute value (these sorts of simple delinearizations are used everywhere in neural nets, and don't require much silicon, so you might as well, as much as possible, just tack them onto the beginning or end of some more substantial processing task, wherever that happens).

More substantial tasks include

- adding a constant (a bias)
- multiplying by a constant (some sort of scaling)
- converting from INT8 to FP16 or vice versa
- simple tensor reshaping, eg transposing, or broadcast (eg convert a 1D array into a 2D array by replicating each value multiple times)

The "Filtering" Circuits can perform filtering as understood in the context of Pooling. So, for example, for 2×2 Max Pooling, first filter will find the largest of each pair of elements in a row and convert the (lets's assume) 8×8 input to a 4×8 output. Then second filter will do the same thing but for pairs of a column, resulting in a final 4×4 output.
Alternatively pairs of values can be added, and the sum weighted in some way (eg divide by two, or four), as in mean pooling.

Post Processor can perform a few specialized tasks, often where not much parallelism is required. For example, based on statistics collected for a plane (first a mean, and then a variance) a standard deviation can be calculated (ie requiring a square root) and then a reciprocal of this, which can be used as a scaling factor for a subsequent normalization pass.

Finally Second Format Converter can do more of the same sort of work as First Format Converter, eg another round of ReLU and suchlike, along with possibly more tensor reshaping.

This covers how Pooling mode can be handled by the two filters. The scheme claims to be flexible enough to handle at least something like 7×7 pooling, though it's possible that anything larger than 2×2 is handled by multiple passes through the Planar Engine?

elementwise mode

In Elementwise mode, simple unary transformations (map each element through a function, square it, or add a bias/multiply by a scaling factor) are obvious.
However binary transformations (eg element-wise addition or multiplication, or select the max from two input planes at each element location) are also possible, and even a basic ternary operation (*elementwise* $A + B \times C$).
We have enough hardware that at least some patterns can be performed in a single pass, for example

$A + B \times C$ might element-wise multiply $B \times C$ in one of the filters, then add A to the output in the second filter.

You can start to see how these can be fitted together. A sum reduction will effectively give us a mean. We can then broadcast that mean (a scalar, or if you prefer, 1×1 array, to act as a full 8×8 array to be subtracted from the original input value. The output of this subtraction can be element-wise squared, then sum-reduced, to give us a variance. And so on.

reduction mode

In Reduction mode, one additional piece of hardware is required, namely a reduction tree, which is provided only within the First Filter.

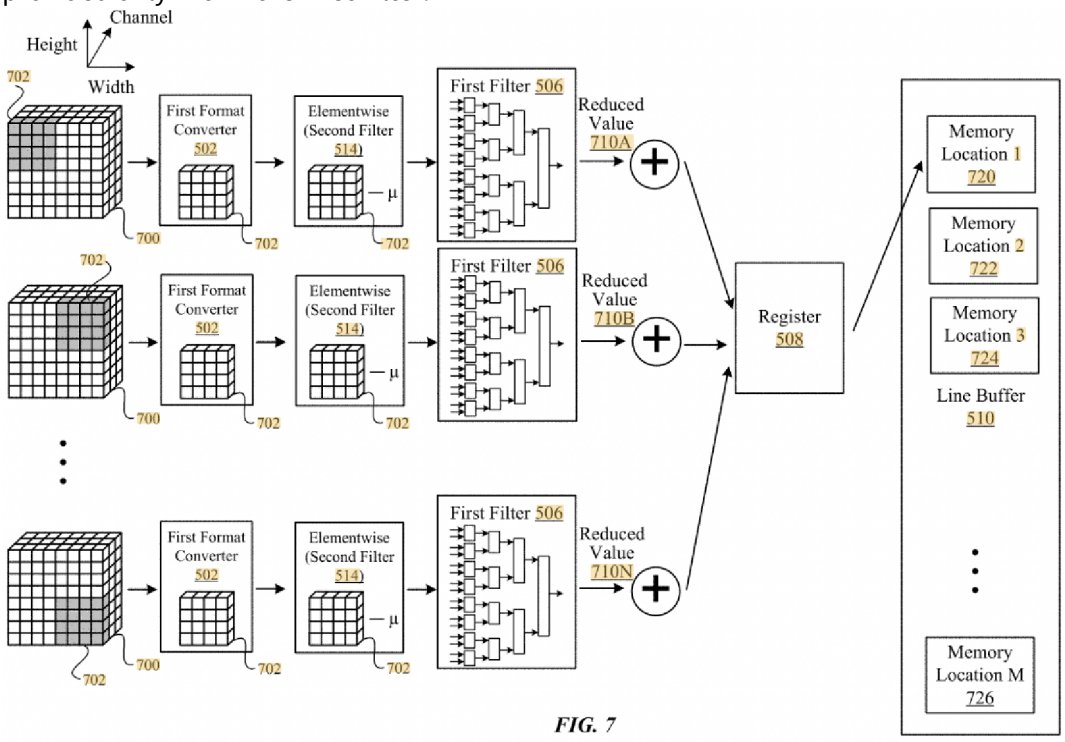


FIG. 7

So suppose we want to find the mean (which begins as the sum of every element) of a very large 3D tensor. We start by splitting the planes within the tensor into units (I'm assuming 8×8 in size) the Planar Engine can handle, and each such unit goes through the Reduction Tree of First Filter. So, as you can see by the diagram, essentially we perform 32 pairwise operations (add, max, whatever), feed the results to 16 pairwise operations, and so-on.

In the particular flow they're showing in this diagram, the suggestion is we're calculating the Standard Deviation. In First Format Converter we subtract off the Mean (previously calculated); in Second Filter we Element-wise square each value; then on a second pass, in First Filter we sum the values through the reduction tree and save the final sum in register 508 which is located within First Filter. Repeat for multiple successive patches. The final value may be stored elsewhere, for example in one element of the "Line Buffer".

We may have, for example, many different channels (recall how our basic vision example resulted in

108 edge channels, then 200 texture channels), and we may want to calculate a standard deviation for each channel, which we can then move out of the Planar Engine (and specifically out of the Line Buffer) as a single vector.

Other variants of reduction mode might be things like, for each plane of a channel we calculate the maximum, so that we gather a vector of channel maxima in the Line Buffer.

In principle we could execute the reduction tree as successive passes through First Filter. But it seems like the arithmetic units are wired up in such a way (the numbers work out, we have 64 units and we use successively 32, 16, 8, 4, 2, 1 adding up to 63 units, then the final 64th unit performing the last sum operation against the value previously stored in register 508) that the first subset of 32 feeds to the next subset of 16, and so on, which means we can pipeline everything processing a patch per cycle flowing through each successive stage.

(2019) https://patents.google.com/patent/US20210158135A1 *Reduction mode of planar engine in neural processor* gives some details about how the reduction mode works (and provides the diagram above showing the Reduction Tree).

So that's the Planar Engine. It should be clear that, while only mildly contributing to the multiply-add capacity of the ANE, it actually provides the possibility of substantial performance improvement for real neural networks by moving many common tasks off the serious compute engines, so that those can execute FMA's every cycle without distraction.

"smart" L2 (producer-consumer flow control)

That's not yet the full story.

The more unexpected additional change in 2019 is what I called the "Smart Buffer" change, described in (2019) https://patents.google.com/patent/US20210132945A1 *Chained Buffers In Neural Network Processor*.

Consider the realities of programming a device like we have described. We might have one (or quite possibly eight) Neural Cores producing data to be consumed by the Planar Engine, or vice versa. Some elements of this are clear, for example the only path between these two halves is via the Data Buffer, so we will need to allocate space(s) within this buffer, perhaps one memory block per Neural Core, or perhaps they can share a memory block, and have the Planar Engine read from this memory. However storage is just one element of the relationship, we also need to co-ordinate timing access to this shared storage.
This is a traditional producer/consumer relationship, but the standard ways such relationships are handled, via locks or polling, leads to a degree of overhead. Just like we offloaded all the indexing, addressing, and looping to rasterizers and shifters, we'd similarly like to offload producer-consumer overhead to a smart buffer.

One part of this is the Flow Control Circuit 332 embedded in the L2 (see the ANE 2.0 overview diagram).
In a sense, this knows the shape (rank, dimensions, layout) of each Dataset defining data generated by a Neural Core or the Planar Engine, along with what elements of this data are present so far in the Buffer. So as enough of the data comes in from a producer, it can be sent out to a consumer without further involvement by other parts of the system. (This block also performs obvious optimizations like using multicasting if the consumer is multiple Neural Cores.) This Flow Control can handle either data flowing from Neural Core(s) to Planar Engine, or vice versa.

In the preferred mode of operation the intermediate buffer is ephemeral, ie data only transits through the L2, to be gathered until enough has been collected for the next step in the process, and then moved to the consumer, at which point the storage can be reused. Either Producer or Consumer can temporarily be stalled if the other side is unable to keep up and data fills up or runs low.
This scheme obviously not only has less overhead of various sorts, but also allows for a fairly lean memory footprint in the L2.

If you think about it, this same model also works for the data being DMA'd into the ANE as a whole, or being sent out of the ANE back to SLC/DRAM. So another job of the Flow Control Circuit is to decide, cycle by cycle, which of all the various data producers and consumers (DMA, Neural Cores, Planar Engine, each possibly reading or writing data) gets control; the patent gives an example of data simultaneously being DMA'd into the Planar Engine, flowing out of Planar into Neural Cores, and out of Neural Cores to be DMA'd back to DRAM; Flow Control Circuit balances all these transfers to try to ensure that no-one ever has to lose cycles waiting.

There are multiple additional technicalities you have to worry about; for example if a context switch comes in, you have to finish up both sides of a Chained Transfer between say the Planar Engine and the Neural Cores before you can actually perform the switch. The patent describes these in detail, but they're not especially interesting.

additional new elements

multi-dimensional (3D) convolution

Next we have (2019) https://patents.google.com/patent/US11475283B2 *Multi dimensional convolution in neural network processor*.
I honestly cannot see the point of this patent. It describes performing 3D convolutions (possibly on multi-channel data) in a way that seems completely obvious given the hardware we've described.

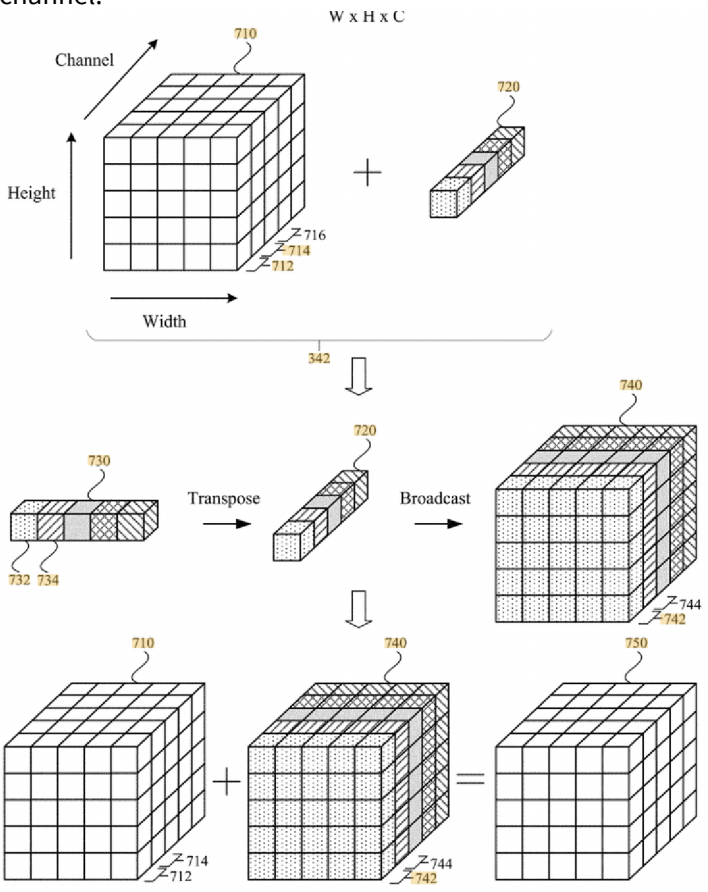
Maybe the point is just that the Kernel Rasterizer and Decompressor of the 2018 design only handled rank-2 kernels (and thus 2D convolutions), and in 2019 someone in Apple requested the obvious extension to make these 3D?

planar engine broadcast mode

Let's just include two diagrams that clarify some of the points already described. The diagrams are basically good enough to give the entire point. These are dated 2020 but are minor updates to the 2019 base Planar Engine patent.

First, from (2020) https://patents.google.com/patent/US11630991B2 *Broadcasting mode of planar engine for neural processor*. ''

So we have some channels of planar data. We want to add constant (an "offset" or a "bias") to each channel.



One way to think of the above (and some other functionality of the ANE) is that ML (at least right now) speaks a language of "tensors" (ie rank-*n* arrays) and so life is much easier for the software (python etc) and then the hardware if you think a little about how to cast your data and your problem into this same language. So, in the above, it's easier to treat a stack of three data planes as a rank-3 "tensor" than as a struct or list, or array of planes, even if those might feel somewhat more natural.

Likewise it's easier to treat the set of offset values to be added as a rank-1 tensor.

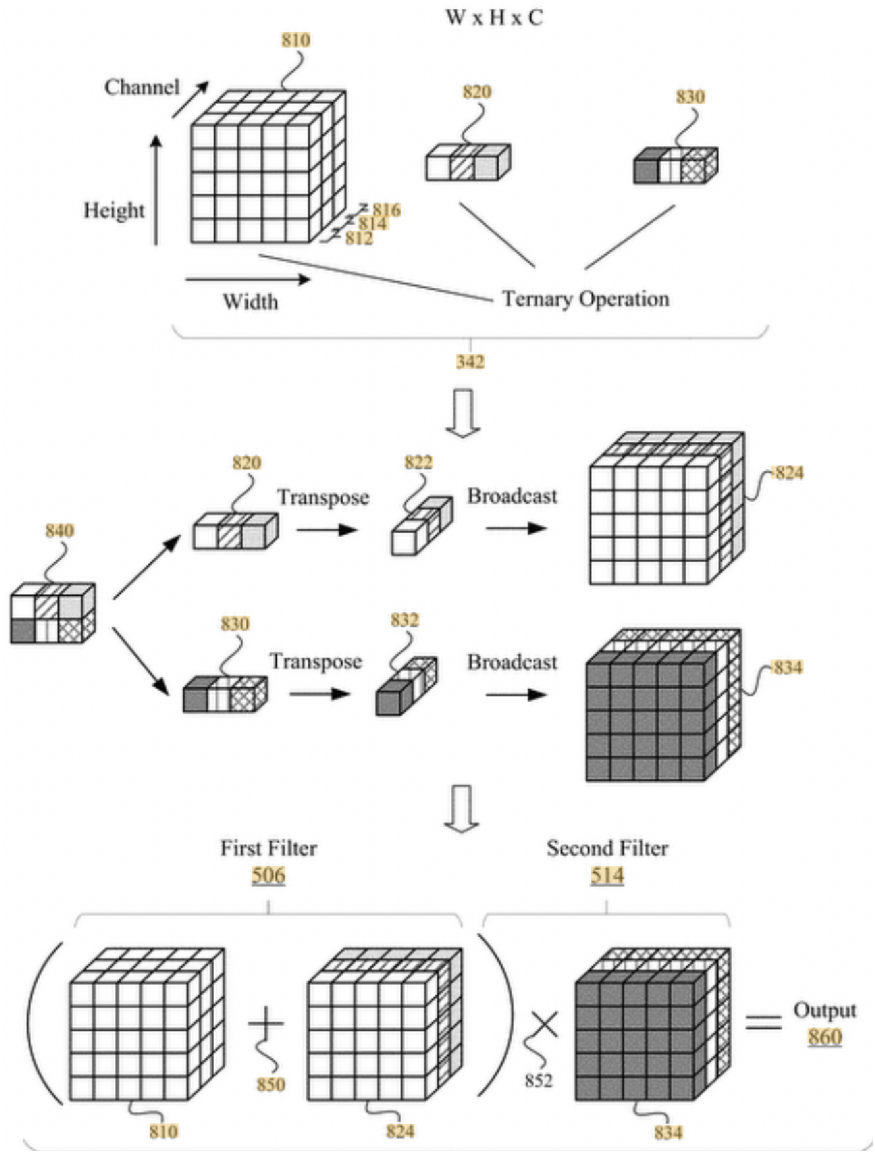
And once you have this uniform vocabulary, you can then implement hardware to handle basic functions (like the broadcast above) and have a fairly smooth mapping all the way from the generic task ("Add this per-channel array to this rank-3 array") and have that implemented with maximal efficiency. Note that, in the above, the "transpose" is more a question of how to program rasterizers

than of data movement. By “transposing” a rank-1 array to be interpreted as a rank-3 1×1×5 array, everything is then lined up for the rasterizers to broadcast values appropriately to be added to each plane as it passes through the Planar Engine.

One way to see where this all might be headed is to look at (2024) https://llvm.org/devmtg/2024-03/slides/Arrays-2.0-LLVM-CGO-2024-Keynote.pdf *Arrays 2.0: Extending The Scope Of The Array Abstraction*, which considers the question of what sort of more advanced and non-obvious things we can do with rank-*n* arrays, given that we are creating so much hardware (and associated languages and compilers) specialized to this particular data structure... It’s worth reading this PDF all the way to the end. We’ll see that some of the (apparently) stranger elements, like sparsity or non-integer indices, will become relevant.

implementation of ternary mode [element-wise (A+B)×C]

Compare this with (2020) https://patents.google.com/patent/US11604975B2 *Ternary mode of planar engine for neural processor*:



The problem of interest now is how do we move the two arguments (the offset to be added and the scale to be multiplied) into the Planar Engine, which is more or less set up for two inputs (in terms of things like rasterizers and tracking data movement into the Planar Engine)? If you listen to what I said above, you re-conceptualize the problem as transferring in not two separate arrays of arguments, but a joint array of arguments, a rank-2 array rather than two rank-1 arrays. Then it's simply a question of minor hardware tweaks to the rasterizer and broadcasting to achieve the desired goal.

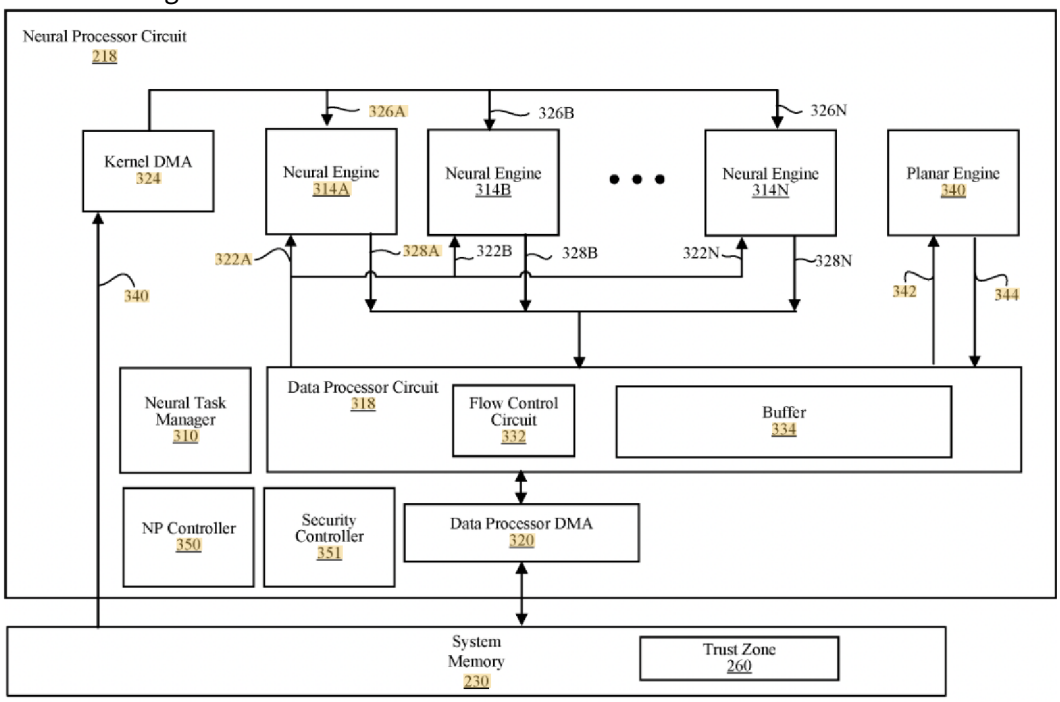
Finally, given the way Apple describe the elements of the Planar Engine, it would *seem* that you could use it, like AMX, as a matrix-matrix multiply engine. Imagine feeding in two 8-element vectors and forming their outer product with Second Filter, then adding it via First Filter to some accumulating value stored in Line Buffer. It seems like something like this could be fitted into the Planar Engine fairly easily.

In some sense this goes against the spirit of the Planar Engine to overload it with this sort of compute-bound operation; on the other hand the overall spirit of the ANE is pragmatism, and this might be a pragmatic way to add more matrix-matrix multiply to the design without adding much more area?

secure mode switching

A necessary but not very interesting update is (2019) https://patents.google.com/patent/US11507702B2 *Secure mode switching in neural processor circuit*. It's not clear how one task within the ANE *could* even spy on another task, but presumably Apple don't want to find out just how clever some hacker might be...

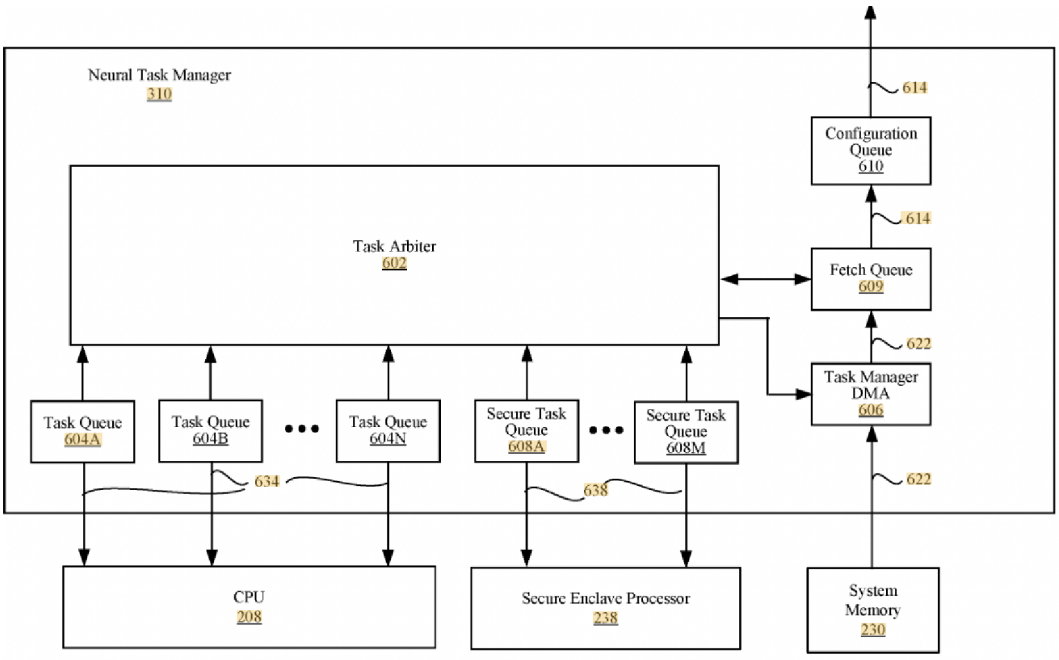
The two changes are:



This is the familiar diagram, only with added elements the Security Controller 351 and the Trust Zone as a secure region of memory.

As before it's probably the case that NP Controller, Neural Task Manager, and Security Controller are essentially elements (software with some hardware) running on the companion core.

Along with this we have

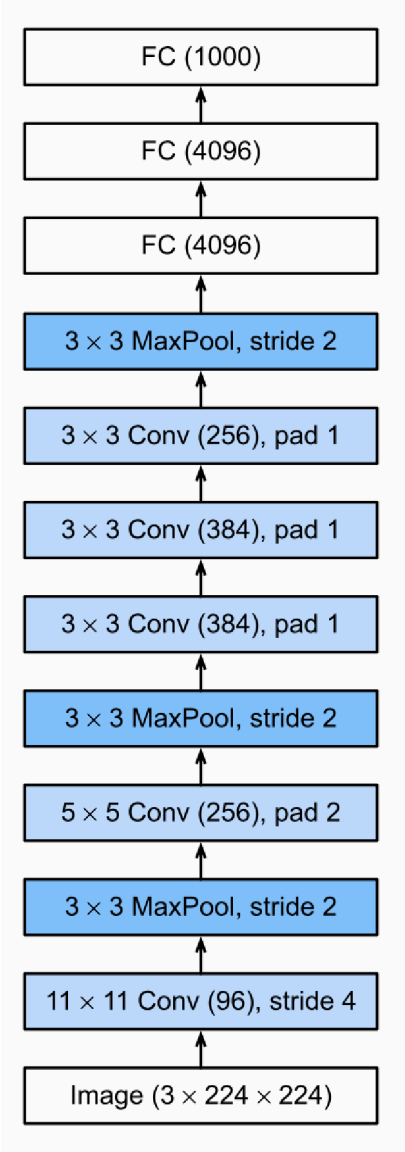


So the basic elements are simple, with no real attempt at performance, just make it basic enough that nothing can go wrong.

Only the Secure Processor can make “secure requests”, not the CPU. When such a request is presented (presumably this means FaceID right now, maybe in future there will be other secure request types?) the ANE is fully drained, ie all current tasks run to completion but nothing new is added, and once no non-secure code is present, it is switched to secure mode, where tasks now are presented by the SEP (and any from the CPU will be rejected). The Secure Tasks likewise run to completion, the ANE is drained, and we switch back to non-secure mode.
From 2019 and going forward these elements are present, but usually not included in the diagrams as they add clutter.

simultaneous independent execution of neural cores and planar engine

Now, think about the Planar Engine as a performance accelerator. How valuable is it, really? So far most of the networks we have seen look like AlexNet,



and the significant thing about this design is that the data flow is serial, one layer after another. For such a neural network, the best sort of performance boost we could hope for is the sort of producer/-consumer relationship we have already described, where we do something like calculate the 11×11 convolutions on the Neural Cores and flow the output from those cores into the Planar Engine for pooling.

Which is nice, such patterns are common, but can we do better?

To some extent we can, because in current neural networks *branching* is common. This does not refer to what CPUs mean by branching, it means that the neural network splits into two (or even more) sequences of layers, than run in parallel until some merge point.

Here's a very simple example.

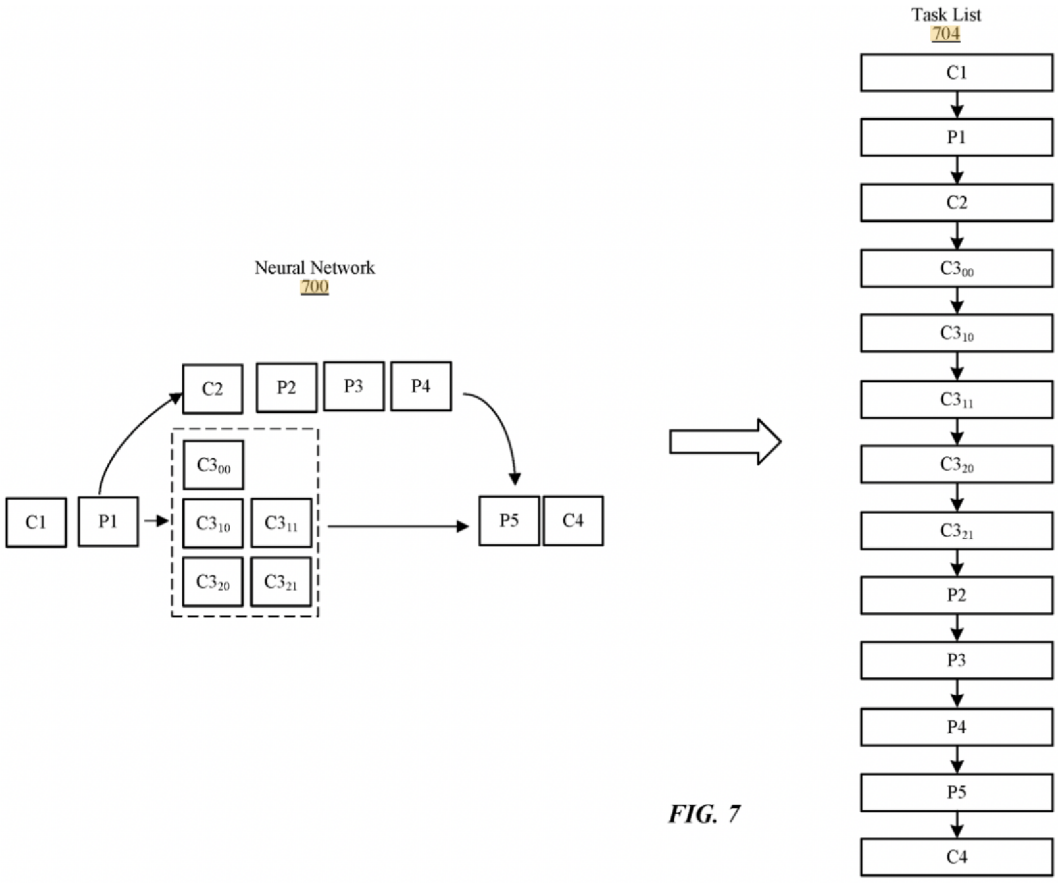
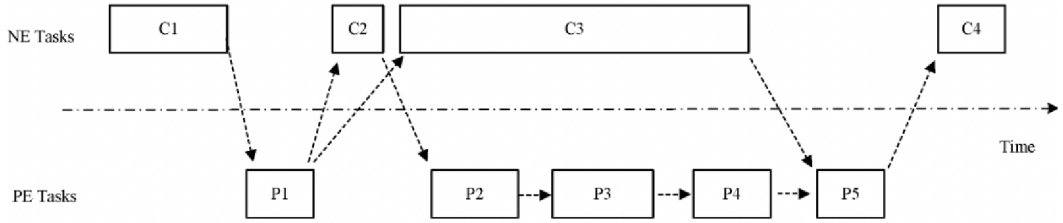


FIG. 7

This is compiled to a linearized layout, but if the linearized layout fully describes the dependencies, then we have the potential to run say P2, P3, and P4 on the Planar Engine in parallel with the set of C3_ convolutions. (The multiple C3_tasks correspond to the compiler splitting up a single complex convolution into multiple smaller tasks). This gives rise to a timeline something like:



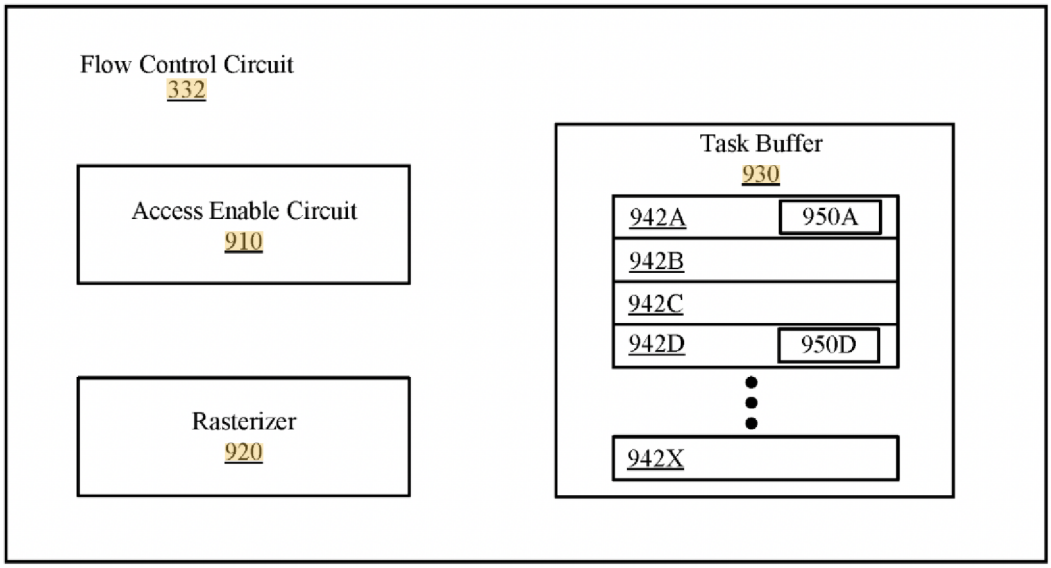
(Ideally we'd then try to pipeline eg C1 to P1 to C2 so that much of the time we'd still have some degree of simultaneous operation.)

The execution of C3 simultaneously with P2, P3, and P4 is a different sort of simultaneous operation of the Planar Engine along with the Neural Cores (no longer producer/consumer), and is described in (2020) https://patents.google.com/patent/US20210271958A1 *Asynchronous task execution for neural processor circuit*.

At the level of Computer Science this is obvious; what's specific to the ANE is how this is done, as part of the fanciness inside the Flow Control Circuit (ie the "Smart L2"). As I've already stated, this Smart L2

knows enough of the layout of each DataSet (ie each input and output tensor) that it can pause or unpause producers and consumers once enough of a DataSet has been generated to provide an input Work Unit for the next piece of hardware. But this smart L2 also knows other patterns of task dependence or independence, meaning that it can also allow (or prevent) the launching of independent tasks from separate branches, as shown above.

The point of the diagram below (explained in a little more detail in the patent) is that based on the Task Buffer (ie knowing which tasks are reading from or writing to which DataSets) and knowing which DataSets are or are not yet complete, decisions can be made by the Flow Control Circuit as to whether a particular task is or is not yet allowed to begin execution.



(This is all neat stuff, but it is, in this design, limited to balancing between two IP blocks, namely the ANE cores as a whole, and the Planar Engine as a whole. There is not yet an attempt to consider fancier, finer granularity options, like running half the ANE cores as producers sending data to the other half of the ANE cores as consumers.)

a technical patent handling precise details of mean pooling

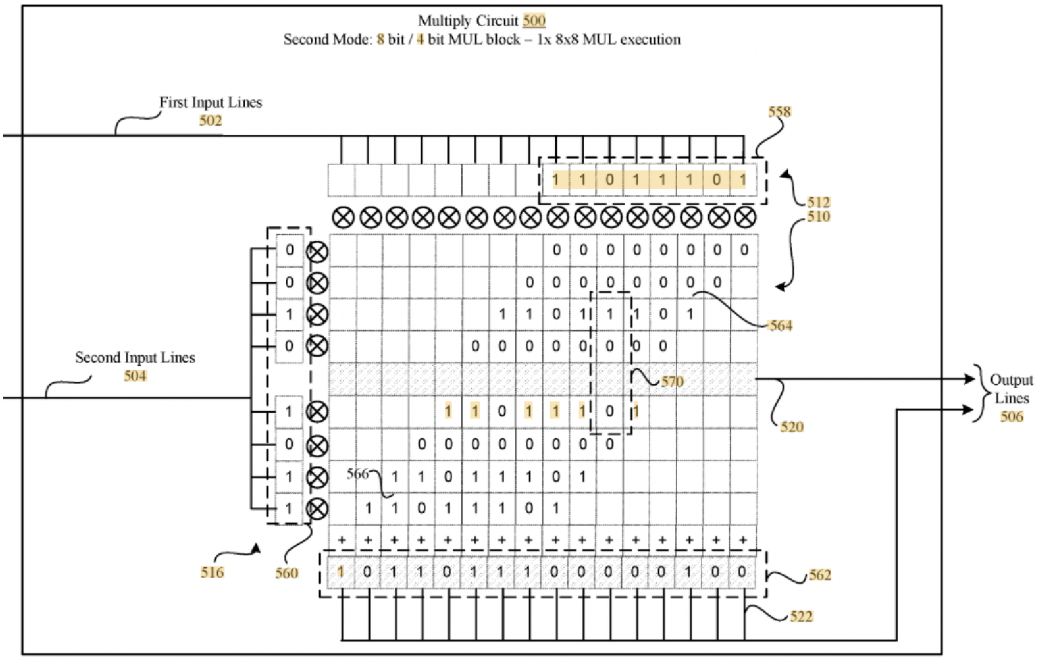
(2020) https://patents.google.com/patent/US11144615B1 *Circuit for performing pooling operation in neural processor* is a rather technical patent, a precursor of stuff we'll see more of going forward.
Suppose you are performing mean pooling, so you're collapsing eg 3×3 patches into a single value representing the mean of the patch. What do you do at edges once a 3×3 block is not available, only a 2×3 or 3×1 block or whatever?
There are many options, but if you're promising mean pooling in hardware, then you have to deliver the mean! So essentially you have to sum up the values in this smaller patch, and then divide by a smaller number (eg divide by 6 or 3, instead of by 9). The patent described one way of handling these edge cases.
Edge cases are, of course, a bane of computing (and algorithm design generally) and we'll see more examples of hardware that tries to deal with them, and thus reduce the burden of the "primary"

hardware having to slow down to handle edge situations.

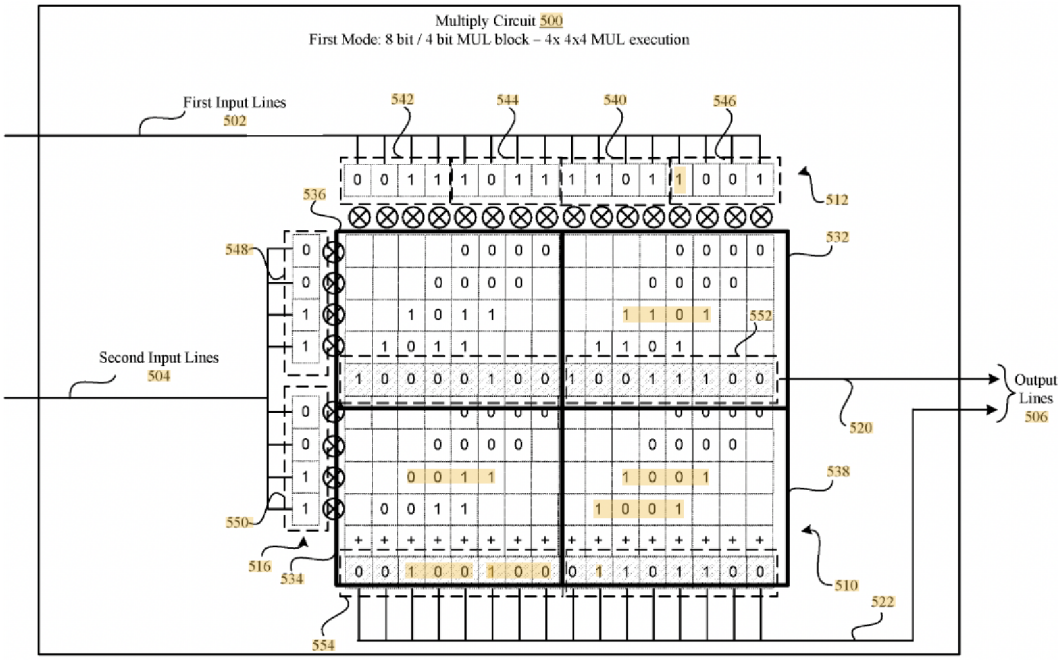
more variable bit width support (4-bit arithmetic)

Finally, the last update for 2019, (2020) https://patents.google.com/patent/US20210279557A1 *Dynamic variable bit width neural processor* is fairly technical in detail, but easy to describe in outline – it's a first pass at providing 4-bit multiplication support.
This is obviously something nVidia keeps pushing with every recent update; let's discuss what is known, then what Apple's version might (or might not) mean.

We've already discussed the essential multiply-add circuits and how (by using a shared multiplier and, if appropriate, faking some values sent into the exponent path) we can in one multiply-adder handle INT8-INT8, FP16-FP16, and mixed multiply-adds.
Now look at the multiplier below.
The important points are that 8bits (so INT8) come in on both paths, and we multiply them. The algorithm should be obvious, it's basically a binary version of manual decimal multiplication. Each bit of the second operand either generates a zero or a shifted value of the first operand, then we add up the results. This is not necessarily the fastest way to multiply if you have long bit strings, but it does the job and may be good enough, especially if you are running at low frequency.



Now consider this variant of the above.



The signal now comes in as 16 bits (to be interpreted four 4-bit values) and what is generated is four INT8=INT4×INT4 values.
The multiplier (the kernel) comes in as what looks like 8 bits, but it's actually two copies of the same four bits.
The upper half of the multiplier multiplies the two middle First Input values, the lower half of the multiplier multiplies the two outer First Input values. (Why this particular split? I'm guessing it was slightly easier to route, but it makes no difference, and could be changed at any time.)
Then it's just a question of performing the sums and routing out the results.

So that's what's definite, we have a scheme to run the multiplier in a four INT8=INT4×INT4 mode.

What is not definite or not even described is various possible extensions to this. Described (but in a vague way, probably not present, at least not in the 2019 design) is extending this to INT2×INT2 multiplication, generating sixteen such results. Clearly an obvious extension (modulo messy wiring/routing); the question is whether it's worth doing.

Next. OK, four values come in, four values come out. But how are they accumulated? Do we have four adders sitting behind this multiplier? And presumably there is some scheme by which these four (INT8) values are stored in four of the eight registers available to each Multiply-Adder in the Accumulator?

Next it's unclear whether mixed precision versions of this are provided. For example there might be a lot of value in a scheme that allows for INT8 signal to be multiplied against INT4 kernel (with two twelve bit outputs per cycle) but something like that is not described.

Finally this scheme could ultimately be fitted into *some sort* of FP8 arithmetic (at least the a version with 3-bit exponent and 4 bit mantissa) just like we can intermix INT18 and FP16 by faking exponents as necessary. However nVidia's two FP8 versions are either E4M3 or E5M2! Now, that doesn't necessarily mean nVidia's versions are better; maybe an E3M4 version is in fact optimal for ML, but nVidia went with what can be fitted into their hardware?
Just like there's no discussion of mixed precision, there is no discussion of any sort of FP8 extension.

All in all this seems like this might be just a "concept" patent, not a "product" patent; a way to note down a good idea for the future, but missing all the remaining elements required to turn it into a product. It doesn't feel tied into the main flow of ANE patents, the way most patents obviously fit together into a single narrative.

It's noteworthy that CoreML (at least as of the 2023 versions) does not seem to support these sorts of very new low bit formats.
But it's **also** interesting to see that they **do** (only in newer chips?) support new format called Linear 8-bit Quantization, described on https://apple.github.io/coremltools/docs-guides/source/quantization-overview.html.
When you see how this works it's clear that it's utilizing much of the technology we have already described, including
- mixed precision arithmetic
- calculating aggregate values (mean and standard deviation) across a tensor and then normalizing
- broadcasting a scalar value (actually ternary operation with two scalars) within the Planar Engine
- one reason why Apple keep mentioning the value of being able to bias and scale a tensor before using it.

You could also see this as something of an Apple equivalent to what nVidia calls the Transformer Engine, namely *dynamic* per-layer scaling so as to allow intermediate layers to operate at lower precision.

Note that the compression we have already described (sparse kernels and kernel lookup tables) only apply to *kernels*. Linear 8-bit Quantization applies to signal, so it's valuable when you are generating large intermediate tensors, large enough that bandwidth moving them between ANE and SLC/-DRAM is worth worrying about.

Also noteworthy is that this Linear 8-bit Quantization support only became available as of the 2023 OS's (eg macOS14 and iOS17). Clearly SW support is lagging somewhat behind what's in the hardware. (Which is not to criticize Apple! It's not easy designing an API to wrap a new type of functionality in a way that will, hopefully, remain relevant as the hardware changes!) So just because we don't currently see ANE support for INT4 or FP8 doesn't necessarily mean it isn't already there in the ANE hardware...

(And, of course, there is the issue already alluded to on that page, of multi-device support. If you add INT4 or FP8 support to CoreML, you are more or less forcing a net to run on the ANE, because GPU and CPU don't support these. Apple may not be ready for doing that until enough of "baseline" neural net functionality is present on the ANE that it's very rare for a net ever to have to exit the ANE and run a layer on CPU or GPU?)

LLMs

So far we've discussed primarily vision-oriented neural networks, and we've seen how ANE 1.0 and 2.0 grow out of that tradition.

At this point, as a refresher, you might want to look at https://en.wikipedia.org/wiki/AlexNet, which has this diagram:

which should now look familiar, with the significance of most of the elements now clear, along with how they map to ANE hardware.

Dropout is a way of preventing a network from overfitting to the training data. I'm not sure that it's relevant to inferencing using the net, as opposed to the training stage of the net. Either way, it's a probabilistic step, which reinforces something I'll be repeating many times below – when we look at the various elements of neural net computing (training and inference) one of the elements that seems like it could usefully be moved to dedicated hardware is various stochastic routines, from the most basic "generate a random bit with probability .5" to the much more sophisticated "draw a random number from this particular distribution".

Another element that's probably more useful for training than inference is *layer normalization*, which is essentially scaling the weights of a layer, after a training step, to a mean of zero and a variance of 1 (something we've seen the Planar Engine can do efficiently). The point of this step is to ensure that the gradients used during each training step stay at much the same magnitude across varying data, and at different layers in the network. A non-normalized layer will have a gradient in the

same *direction* as a normalized layer, but due to specifics of the (non-normalized) layer, may have a much larger magnitude than the gradient of some other layer, which can skew the overall movement of the network weights so as to decrease the penalty function.

Now, vision is interesting, but obviously the hot new thing is LLMs, which have somewhat different technologies that one might want to implement on somewhat different hardware...
Probably the best overview of this (technical, but not too technical) is the one written by Stephen Wolfram, https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/
Read that now (it may take you a few days, don't skip anything!) then return to this PDF.

Let's note a few take-aways from Wolfram's article, in terms of basic operations and data structures we now care about.
- one element of running an LLM is drawing a random number from a particular distribution.
The "zero temperature" model at each stage generates a list of (possible word, probability=p) pairs. But we don't want to run the model selecting the highest-p word at each point. Instead we want to transform this list into something like a Boltzmann distribution (probability of any element is proportional to $e^{-\beta p}$) and then draw from this distribution.
This gives rise to both engineering research questions (is the optimal distribution Boltzmann [which tends to exaggerate the differences between elements with different p's] or a "flat" distribution [based directly on the p's] or something that's Boltzmann like but much cheaper to compute (eg a quadratic or cubic polynomial in p)? And EE engineering questions, like what's a cheap way to draw a value from such a theoretical distribution? And how many elements in the distribution do we need to care about? Can we limit ourselves to the top 32 and ignore the rest?

- training a model (and the very construction of a model) consists of multiple layers of different types. In many cases we have some idea what a layer is doing (for example ReLU, Tanh, and Sigmoid all do "somewhat" the same sort of thing, and you could imagine a "capped ReLU" that clips values greater than 1 to 1, to be even more similar to Tanh and Sigmoid). Do the exact differences between these matter? Or can we get away with using the cheapest one, in terms of transistors, especially if we train to this cheapest one?
This is somewhere where I expect Apple (and nVidia) to have a substantial advantage over people who are just grabbing academic networks from Hugging Face and trying to repurpose them, without ever thinking clearly about quite why each layer on the network is there, and how to optimize that layer.
This may sound trivial and obvious, but consider now the softmax layer in may LLM (and other) networks (for example softmax is the last layer of the toy digit recognition network Wolfram discusses in his post). Softmax takes a vector v and outputs a vector of the same length; specifically it calculates $\frac{e^v}{Z}$ where $Z = \Sigma_i e^{v_i}$. So for each element of the vector, v_i, calculate the exponential of this value along with the sum of all these exponentials, and normalize by the sum.
Now instead of just glossing over that, think! What's the point of, what's it doing? Essentially what we want to do is, from a feature vector, discover the "most important" features. Being precise, we want to

do two things.
+ first exaggerate the features with the largest weight relative to features with less weight (the exponential does this, “stretching out” large values even larger and “shrinking down” small values even smaller.
+ then normalize, so that the features with the largest values stand out (close to 1) and everything else is close to 0.
So, for example:

```
In[•]:= softmax = SoftmaxLayer["Input" → {5}];
        softmax[{0.1, 4.5, -0.2, 3.3, 5.4}]
Out[•]=
        {0.00324611, 0.264398, 0.00240478, 0.0796353, 0.650315}
```

Once we understand that this is the goal (find the *indices* of the *few* features that are *strongly present* in the vector) multiple alternatives open up as research options.
Perhaps we can stretch and shrink the inputs in some way (again, perhaps, a simple polynomial?) rather than the full cost of an exponential? Do we need the normalization if we can just find the *N*-largest values after stretching?
And when we propagate this layer to the next layer, do we actually need the floating point softmax values? Perhaps we can get away with collapsing almost all the values to 0, and marking the non-zero values as placed in one of four or sixteen discrete bins?

There appears to be a lot of scope for careful redesign optimization of existing networks in this way, but it’s co-design between the network functionality and the provision of particular logic circuits that seems barely explored so far.

- Step one of an LLM is convert the next token (think a number between say 0 and 50,000 representing a word, or word fragment) into an “embedding”, a vector of say 768 numbers (for chatGPT) where the value of each of the 768 numbers represents, in some vague sense, the amount the word represents one of 768 “semantic dimensions”. These 768 element vectors are then manipulated in various ways (essentially through a model that’s predicting “based on all the text I have ever seen, what are some reasonable guesses for the next 768-dimensional vector I would see given this most recent token and its predecessor stream of tokens [ie a stream of 768-dimensional vectors”.
At this point a few things should be already be clear.

+ How cool is this? Semantic dimensions! So much to study and investigate here. For example do different languages use essentially the same semantic dimensions, so that if I pre-train on Russian, then use what I learned, like the appropriate embeddings, for English, is my model already 90% trained? How do these embeddings evolve if I train successively on text aimed at first graders, second graders, third graders, and so on? etc etc.

+ Note how clever this embedding idea is! If you design a word prediction engine based on words (so

list each word, and number them 1..50,000) you have to get the prediction exactly correct, because word 2731 (regardless of how you ordered them) is different fropm words 2730 and 2732. But by embedding, “all I have to do” is predict a location in 768-dimensional space, then find the nearest word (or more likely set of words, then choose one of them). Point is, predicting somewhere near a location using real numbers is a lot less fragile than predicting an exact integer.
Note also this means another piece of dedicated hardware we may want in our LLM machine (along with drawing a random number from particular distributions) is something that can find the *N* nearest points to a point in some high dimensional space.

+ How much genuine information is there in these embeddings (and then the various matrices that will multiply them as the starting sentence propagates through the neural network. Can I, for example, collapse this 768-element vector representing a particular token to something like a set of 16 indices (the only non-zero values) and for each index a 4-bit strength? Similarly are most of the matrices that manipulate these vectors essentially empty?
I still have not been able to find a good answer to this question! The fact that nVidia sells its sparsity hardware (50% sparse matrices) gives an upper bound; and the fact that pretty much every model runs just fine on some sort of high-end mac using 4-bit quantization or so (and that applied by people who are really just trying blindly, they generally don’t know much about the target hardware or the model structure) suggests upper bounds. When I generate a histogram of the model weights for a layer of GPT-2 the histogram looks Gaussian. If we assume that the values close to zero essentially cancel each other out, so all the real value is in elements 1 standard deviation or more from 0, does that mean we can throw away about 2/3rds of the values?
Another way to put this is people talk about these model sizes as, say, 70B. But is that 70B a marketing number (puffed up to make it sound impressive and intimidating) or is it an engineering number, eg the smallest number of bytes into which you could reasonably compress the model using a sensible and appropriate lossy compression scheme?

A reason I keep stressing degree of sparsity is that the degree of sparsity affects the optimal hardware for the problem.
Suppose both matrices are dense. Then AMX-type matrix-matrix-multiplication hardware is probably optimal.
Suppose the degree of sparsity is ≳50%. Then nVidia-type hardware is probably optimal.

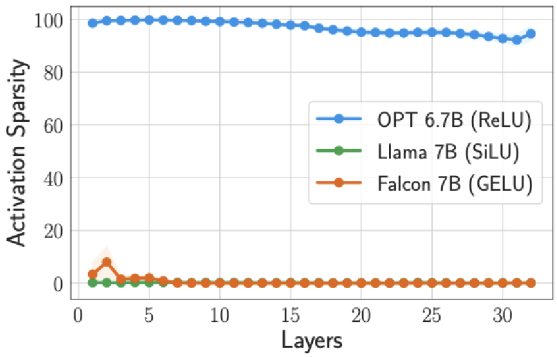
But suppose the density of non-zero weights and signal elements is around 10%. Then only about 1% of products in the dot-product of two vectors is non-zero. In such a case, you want very different hardware. Probably something that
- encodes each column or row of a matrix as an array of (skip-count of zero’s, next non-zero value)
- performs matrix multiplication as a collection of dot products
- has a dot-product unit which loops loading these (skip-count of zero’s, next non-zero value) tuples for the two vectors going into the dot product, mainly crunching through memory loads until we get two matching offsets in the two vectors that are both non-zero. (So the hardware is mostly multiple

hardware counters tracking offsets into arrays, and MSHR's tracking memory requests; all hooked up to a usually sleeping single multiply-accumulator.)

With the above as background, look at this paper (by a bunch of Apple people) (2023) https://arxiv.org/pdf/2310.04564 *ReLU Strikes Back:*
Exploiting Activation Sparsity in Large Language Models. The paper makes two primary points.

The first is that ReLU is a great delinearization function for hardware because it is so simple, but that simplicity comes with few downsides. More complex ("smoother") delinearization functions provide slightly faster convergence or accuracy, but probably not enough to justify their increased computational cost.

The second is that ReLU, as a consequence of its aggressive clamping on the the negative side, generates activations that are 90% sparse (ie ≲10% of weights are non-zero). Which, as I've said, has important consequences for weight storage and optimal dot product hardware. (Even on multiply hardware that's not especially optimized for sparsity, this still gives you notable computation savings. This is hardware that can only drop the dot-product computation if an entire row or column [or more precisely, a segment of the matrix-multiply blocking of a row or column] is zero.)



(a) Sparsity of different models

- Another interesting aspect about these existing LLMs is that they are "flow-through": data enters at one end and comes out the other after a fairly predictable amount of time (with one more token added to the sentence). There is no backward branching within one execution of the entire neural net.

One obvious question is: "to what extent is the human brain the same way"?

Another point is that, to some extent, this is what is being worked around by changing an LLM request into something like "<do such and such>. Please think though the problem step by step and list your thinking at each step".
This enforces some sort of backward looping (and may play the same role when a careful human thinker goes through a problem step by step rather than leaping to intuition).
But enforcing backward looping at this outermost level may be computationally very inefficient. Perhaps we could do a lot better by allowing looping deeper within the network? (At the cost of now being subject to the Halting Theorem... You may now ask some questions and have your answer take some indefinite, up to "infinitely long", amount to be formulated...)

ANE 3.0 and later

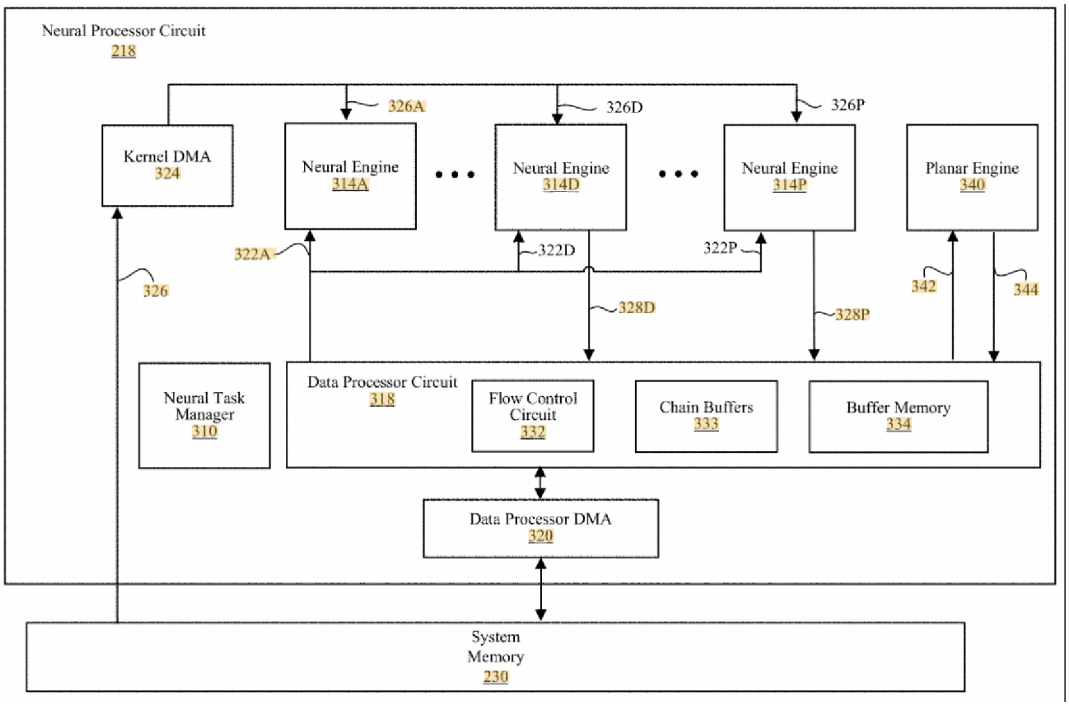
So now we enter the 2020 update. 2020 is the year of A14 and M1, and the patents seem to match that sort of hardware (though, as always, some patents may be aspirational, or may have proved difficult to implement and so only get exposed a year or two later).
The A14/M1 ANE has an obvious big update in that we double the number of Neural Cores to 16, but most of the changes have to do with more subtle functionality to get more out of those cores, fewer dead cycle while the cores are not running at 100%.

There don't yet seem to be any large changes to the ANE like the 1.0 and 2.0 designs, rather various small (but useful) changes, many of which seem perhaps more on the side of helping with training, not just inference?
For this reason, I'll now follow changes by functional area, rather than grouping all the changes of a single year together.

Task Sequencing and Scheduling

(2020) group cores into 4-long chains (just topology? possible first step towards communal action?)

The most significant "structural" change is (2020) https://patents.google.com/patent/US20220036163A1 *Chained neural engine write-back architecture*. The overview picture is



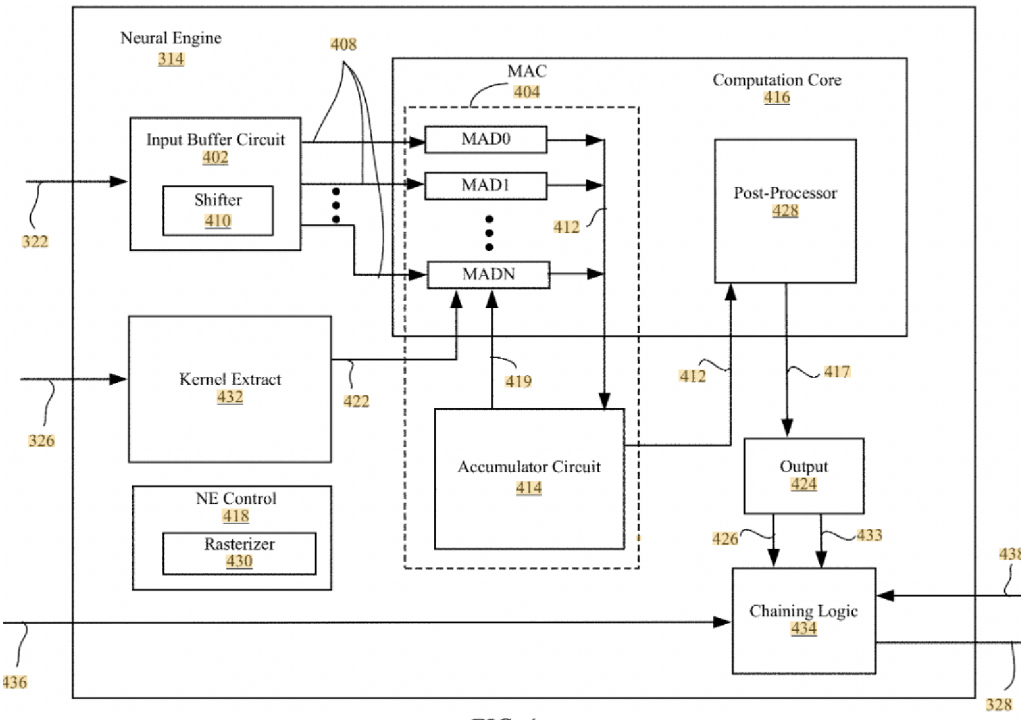
and we don't seem to see much change, except for Chain Buffers 333. But these Chain Buffers begin

the process of opening up new ways to use the device. The part that's actually patented seems really trivial, the idea of a buffer that accumulates small data blocks into a larger data block. Much more interesting is the background description, and what it suggests.

In the 2019 model we saw a very limited way to automate synchronization between parts of the ANE, for example with the Neural Cores acting as producer to the Planar Engine's consumer, and with the Smart L2 ensuring that the two sides didn't overflow or underflow, while avoiding the overhead of locks or polling or suchlike. But, as I pointed out, this was a very limited model only capable of synching "all Cores" against "Planar Engine".
This 2020 update removes some of this limitation, and seems to allow you to split the pool of 16 Neural Cores into subsets, each working on different tasks. (Or if not exactly that, a step on the way to such independent activity.)

We define a block of Neural Cores as a "Chain" (which you can think of as a synonym for cluster) and the Chain Buffers consist of some amount of buffer SRAM, together with some registers defining the clients of this Chain Buffer and how they synchronize against each other (eg as producer/consumer). The patent seems to suggest an ultimate goal of flexibility in how chains are define, while also suggesting that, as of this design 3.0, chains are fairly hard-wired into there being four of them, each four cores in size.

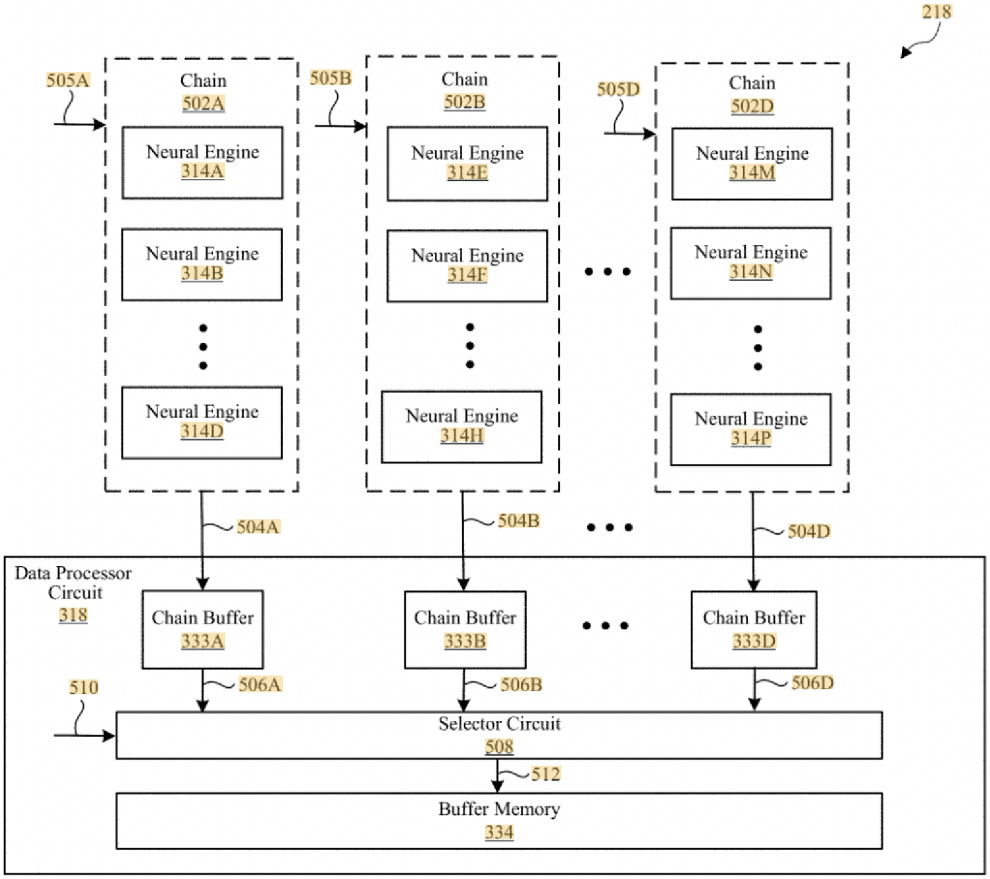
If we look at the per-core level we have:



The new item is the Chaining Logic which provides, at a per core-level, a way to throttle output, or to route output into a successor core (hence the logic of the term "chain" for a group of co-operating cores).

Recall that Output 417 has always been able to resequence and reformat data; but now it's perhaps easier to understand the point; that it can massage the data out of Core *N* into what's desired for the layer being performed by the next core *N* + 1 in the chain. Output 417 also now gets an additional output line, namely line 433, which is a flow control line controlling when Chaining Logic 434 passes data on to the next Core in the chain.
So, overall, the chaining scheme allows a core in a chain to feed data into the next core in the chain without going through the L2, and with minimal delays between cores waiting for data. This is a nice energy saving relative to the 2.0 design, where all communication between cores had to go through a buffer in the L2. **IF** it used that way... In this first design, it does **not** appear to be used that way; chaining is purely about topology, about reducing and reusing wires. But one day...

For the ANE as a whole, we now have something like below, which shows the 16 Neural Cores split into four chains, each working on some task.
Don't be fooled: the diagram makes it look like data is flow flowing from the first chain (cores A, B, C, D) to the next chain (E, F, G, H) but the arrows 505A, 505B, .. 505D are actually *control* signals to the chain; the data flow is down the chain, exiting the chain into the Chain Buffer within the Data Processor Circuit (ie the "Smart L2").



First step in the process is output from the chains is aggregated into the Chain Buffers. Any output smaller than 256B is aggregated to 256B before moving on. Presumably operating with these larger

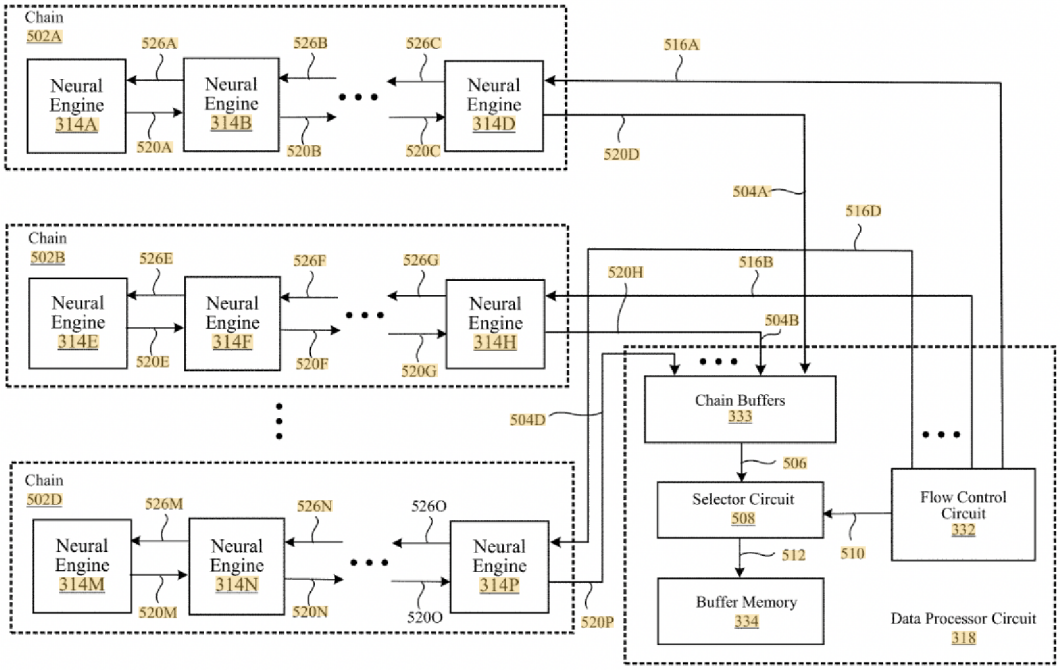
chunks helps reduce tracking overhead.

As a minor detail, the patent seems to suggest that the data width of channel 504A (and the others) is about 64B.

The topology seems very simple in this 3.0 version, not even a ring, just four chains of one-way data flow, terminating in the Chain Buffer storage.

Then an arbiter (Selector Circuit 508) decides on a cycle by cycle basis which of the Chain Buffers A..D gets to pass on its aggregated payload of 256B into the L2 SRAM.

Here's a view of the same thing in a little more detail:
Once again the data flow is the 520 arrows. The 526 arrows are control flow as each upstream element indicates that it does (or does not) have space to accept new data.



The most limited interpretation of this design is that four elements of a chain can work together on one task (eg all applying different kernels to the same Work Unit; or applying the same kernel to different work units from the same tile); but in a way such that each core generates data which flows down the chain till it reaches the Chain Buffer.

A more capacious interpretation would be that data can flow from one core to the signal input of the next core, so that you can set up the four cores as a task pipeline.

It's unclear which of these interpretations is correct, but it *seems* to be the first. I found nothing to suggest that data can flow out of the "Chaining Logic" [which is basically buffering and routing] of one core to the input of another; the only option seems to be to flow down the chain till the data hits the Chain Buffer.

In other words, if we take the patent literally, what it describes is a particular way of wiring up 4×4 cores into four chains (with very simple routing from one core to the next in the chain) so that output data can be aggregated at each Chain Buffer, and then large amounts of data written in one cycle into the Smart L2. It's probable that this idea (routing and common chain buffer) is all that was

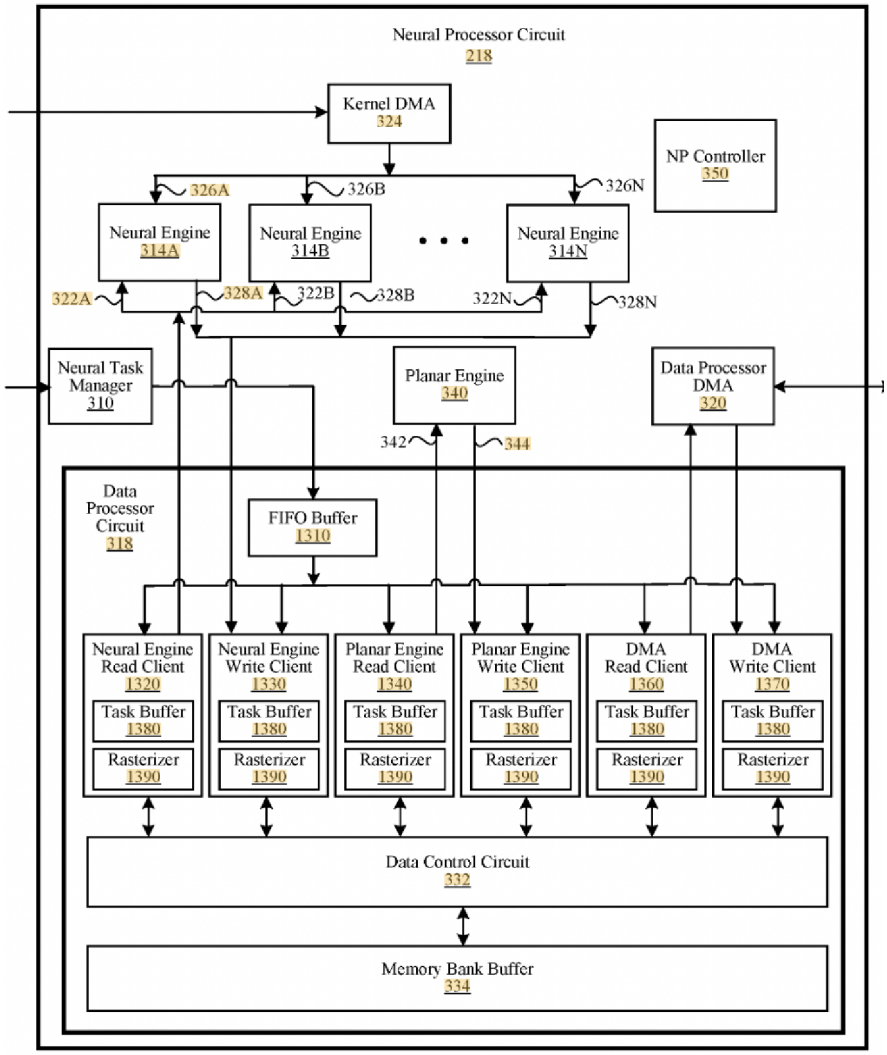
implemented in the first version of this idea; but it certainly seems like it's setting things up for some sort of tighter co-operation and core-to-core data flow within a chain.

Another obvious question is how does data get to the input of each core, as opposed to the output? Perhaps there is a second, parallel, network running in reverse, so that again a single L2 transaction can deposit 256B of data into a Chain Buffer, which then feeds that out 64B at a time moving up the chain to the target core?

(2020) better scheduling of various independent ANE tasks

Next up is (2020) https://patents.google.com/patent/US20220036158A1 *Task skew management for neural processor circuit.*

In principle we have multiple queues storing different task descriptors, and dependency data, so we could just let the system run as the dependencies allow. Ideally we would just fire off tasks (DMA, Planar Engine, maybe even separate tasks to independent distinct Cores) as dependencies allow. This diagram shows the idea



We see that tasks flow into the Neural Task Manager 310 thence to a FIFO buffer, which distributes them to one of six clients. A client is essentially a list of task descriptors (up to around 12 or so per client) along with dependency data. Each element in the Task Buffer is a stub that refers to the full Task Descriptor in the L2 SRAM.

However the above scheme may be too aggressive. We do not, in fact, want the skew (ie the gap between the furthest task in the task descriptor list being worked on and the nearest task in the task descriptor list being worked on) to grow too large.

Why not? The patent suggests that this generates too much complexity in terms of possible dependency interactions.

Another way to say this is that one might start burning additional power trying to track, in each cycle, which of multiple various things can happen next.

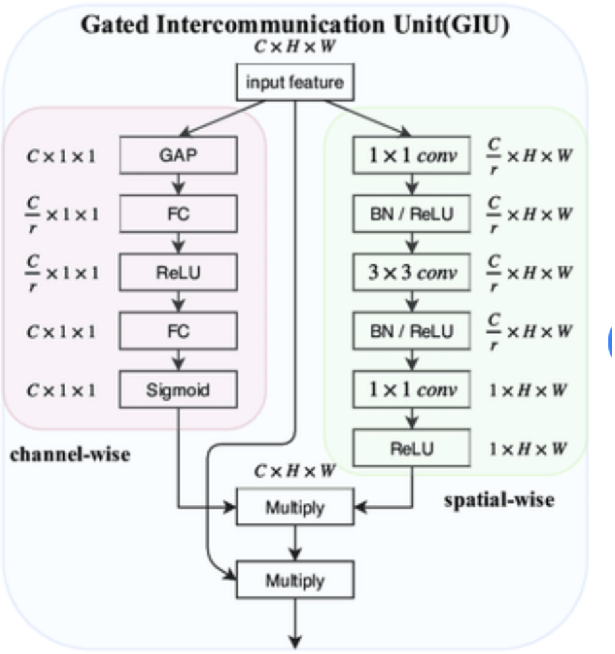
So bottom line is: we wish to limit this skew. The patent describes a particular way of doing this (which has, as a consequence, that even though it looks like the system can be buffering and, in some sense partially executing simultaneously, 6 queues × 12 slots per queue =72 task descriptors, in fact

the system can actually only be buffering and, in sense partially executing, a maximum of something like 12 task descriptors).
Which still seems reasonable in terms of trying to overlap all the different compute engines.

(2021) allow data-dependent ("conditional") branching

Continuing the 2020 theme of adding strange elements we didn't know might matter for neural networks, look at (2021) https://patents.google.com/patent/US20220237439A1 *Branching operation for neural processor circuit*.

Branching in this context is somewhat ambiguous. Many networks have a structure like this below, generally referred to as *branching*.
This sort of structure can be exploited to improve performance (we've already discussed how it might allow for overlapping Core and Planar Engine tasks).



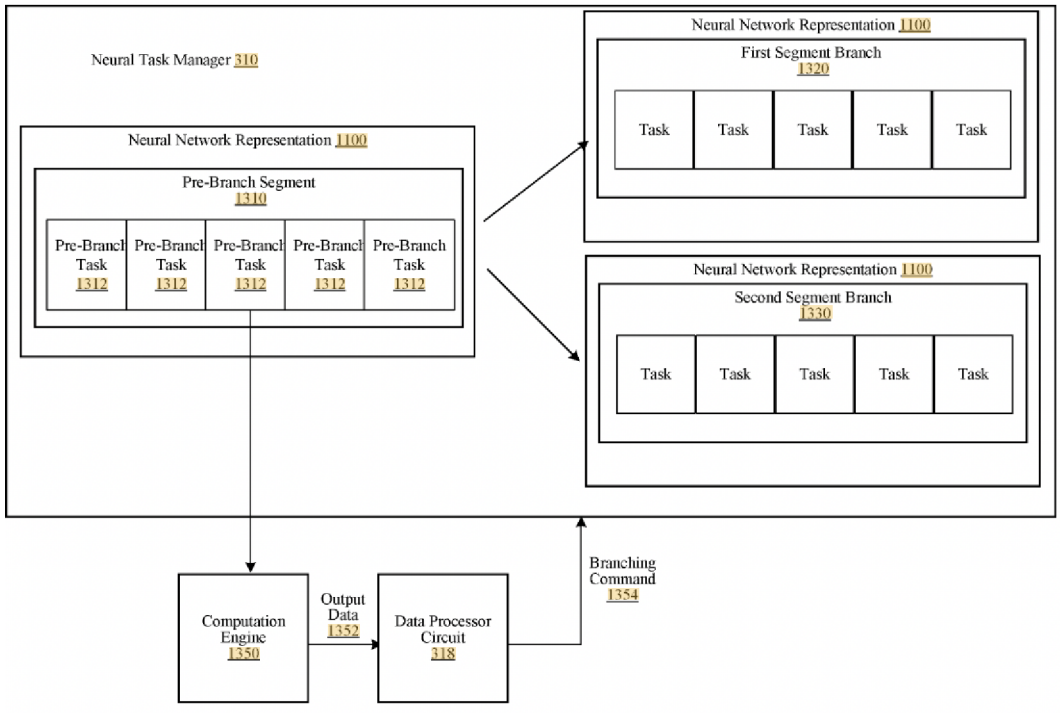
But this structure still part of the static flow of the code; it's not what a CPU person would consider any sort of conditional branch. The patent is about bringing *conditional branches* to the ANE.

So why is that useful? This is still being explored by researchers, but one answer is specialization allows for optimization. You could, for example, have your first pass recognition system detect that an image contains a human, and then conditionally swap in a face recognition net, otherwise if the image contains a dog, you might swap in a dog breed recognition network. Or the first stage of a network might recognize a language, then pull in a specialized recognition network for that language. And so on.
The point is coarse-grained branching, choosing between one network and another; it's not fine-

grained decisions within a network layer or (not yet...) looping backwards to, eg, re-execute a layer multiple times.

The fact that branching is supported is interesting; nothing else about the patent is! The compiler splits the net into elements called segments, sequences of layers, which are essentially “things that can be branched to”.
The start of a segment gives some indication of what will be tested (presumably some scalar value earlier generated within the ANE) but we get no indication as to how this testing is performed. Based on the test result, we then either route next to segment A or to segment B.
The idea seems to be, as best I can tell, that to some extent the context switch mechanism is over-loaded to perform the branching. That is, we start loading both possible target segments, like we expect both to be executed, then the result of the comparison either lets us flow down the default path (while flushing the other path), something like the CPU equivalent of not taking a branch; or the result compels the Neural Task Manager to do something that looks somewhat like a context switch, we flush the default segment that was being prepared for execution and “context switch” to the other segment.

We can see the idea, and the timing here. A task within an earlier layer ultimately generates a branch result which flows out the computation core to the smart L2 (ie Data Processor Circuit 318) to the Neural Task Manager 310 (ie to the ANE Companion Core). Based on the branch result, the companion core then fiddles with the tasks queued up in the task queues so that we route to the tasks from one of the Segments and not the other. All this takes time, so we perform our branch test as early as possible, hopefully with lots of additional subsequent work to be performed by the subsequent Pre-Branch layers.



Given this idea you can start to imagine a few issues.

First do we want to delay executing the later segment if the branch result is not ready in time? Of course not. So there's the potential for at least some degree of speculative execution (for example you can start to prefetch and configure everything assuming one of the segments; and then flush it [faked as a Context Switch] if you guessed incorrectly).

Second this makes genuine context switching more complex because there's more state to be tracked; and we have to make sure we do the right thing if we plan to context switch partway between the execution of a branch comparison within the ANE and the consequent re-sequencing of tasks and segments within the Neural Task Manager.

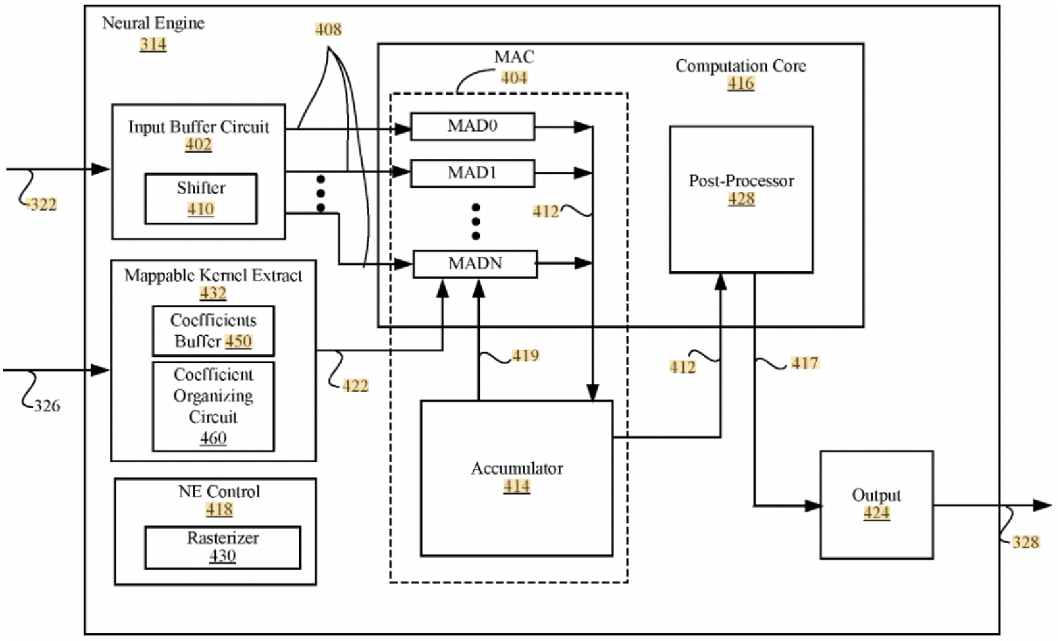
(2021) slightly more efficient context switching

By the way, since the context switching was being revised anyway, it was also made a little more efficient. The Neural Network Representation (basically the binary fed to the ANE) now has a way to specify which temporary/intermediate Datasets within the L2 have become irrelevant, as the computation proceeds, so that a context switch can avoid copying out those regions of memory.
The details are covered in (2021) https://patents.google.com/patent/US20220237438A1 *Task context switch for neural processor circuit*.

Kernel Handling

(2020) kernel mirroring and rotation

First up, we have a change to how kernels are handled, in (2020) https://patents.google.com/patent/US20220108155A1 *Mappable filter for neural processor circuit*. The Core diagram now looks like



and the obvious kernel change is the addition of Coefficient Organizing Circuit 460.

The problem we want to solve is dealing with models that are too strongly trained against data presented in a particular way. For example, even though people may struggle to find the dog, or recognize a face, when an image is turned 90 degrees or upside down, we want our computers to do this and do it better than humans!

There are many ways you might imagine dealing with this, starting with rotating the image. But the particular scheme patented here has the new Coefficient Organizing Circuit 460 read out the kernel coefficients(eg a 3×3 edge detection kernel) rotated by say 90 degrees.

This seems extremely specialized, something like a quick hack added to support FaceID on the iPad (which allows FaceID login in both portrait and landscape orientation). But read on...

The behavior is like:

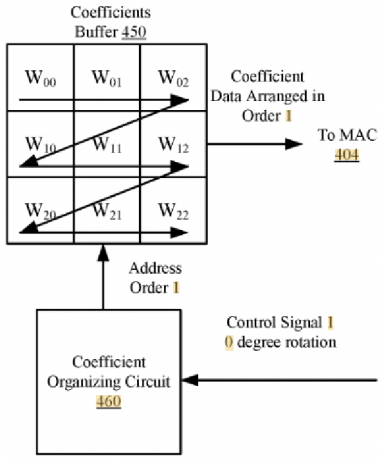


FIG. 6A

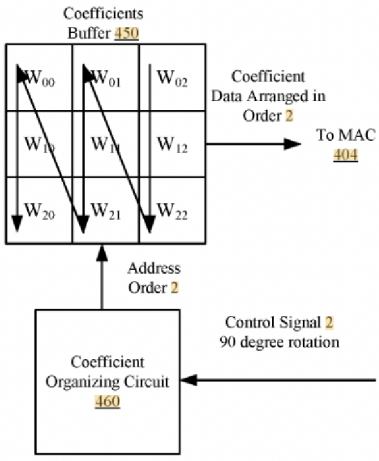
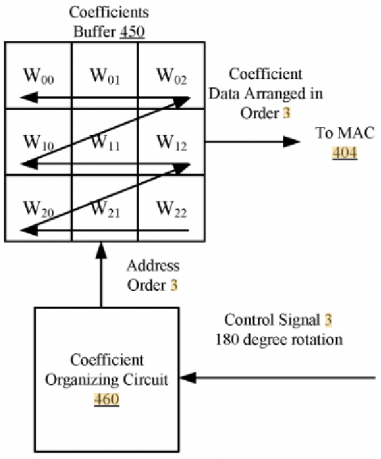
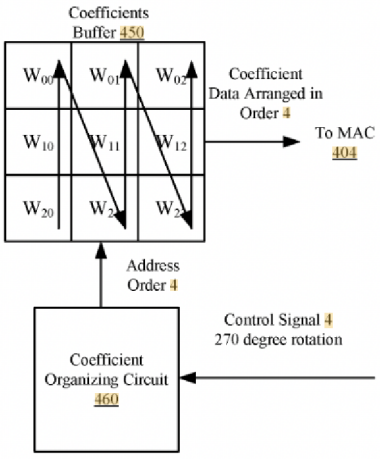


FIG. 6B





This can also be used in training, and that becomes more interesting. Instead of four sets of edge detectors at different orientations, we can use one set, and rotate it three times during training and then during inference. The patent claims that building nets based on geometrically transformed kernels rather than transformed input images, results in both faster training and smaller models.

Another way to view this is as an additional form of kernel compression. For example the set of 108 kernels used in the first (line detection) stage of our example vision neural net might collapse to 27 kernels each rotated/mirrored four ways?

At first this might seem vision-specific, but maybe it's generally useful for ML detecting something in the physical world? If the signal has some sort of 1D, 2D, or 3D symmetry, maybe there is a portion of the set of kernels amenable to this sort of compression? For example in working with audio, convolution kernels may be used that vary in time, and reversing those kernels might have some value?

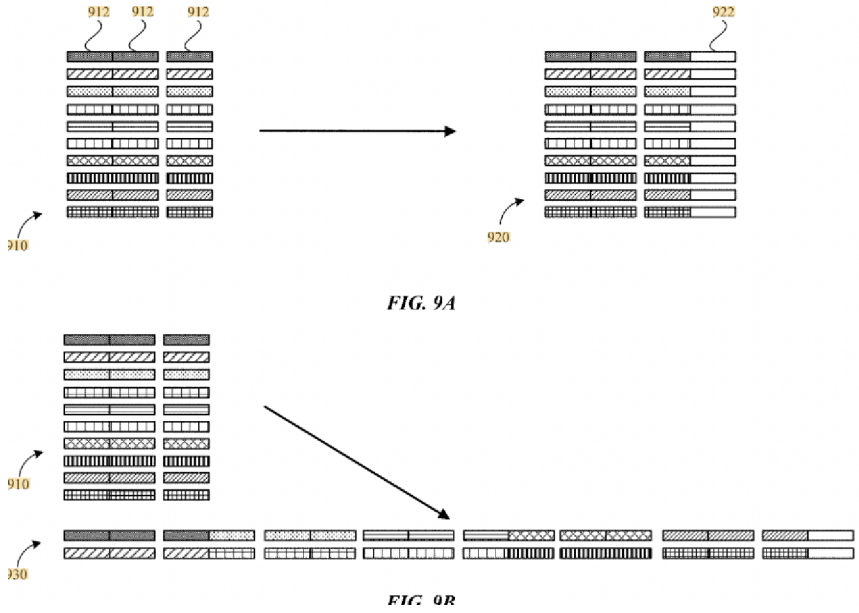
Once we go down this path of "plausible ways to compress a kernel" additional options open up. The re-arranger circuit, which programs these alternative paths through the kernel coefficients can also be programmed for mirror reversal, or even scaling along one or more dimensions (to transform say a 3×3 kernel to a 5×5 kernel.

Which makes you wonder how small, eg, that set of 108 AlexNet edge detector kernels can be shrunk...

(2021) handling non-power-of-two work units

Something you have surely noticed when looking at the classic networks like LeNet or AlexNet is that the intermediate stages use weird array sizes like 28×28 or 55×55. At some point enough engineering and experimentation will probably be thrown at these to fit them into more computer-optimized sizes (with some form of edge padding); but for now we are stuck with these. How can we fit these into our 256-element Core design as effectively as possible? This is the subject of (2021) https://patents.google.-com/patent/US20220222509A1 *Processing non-power-of-two work unit in neural processor circuit*.

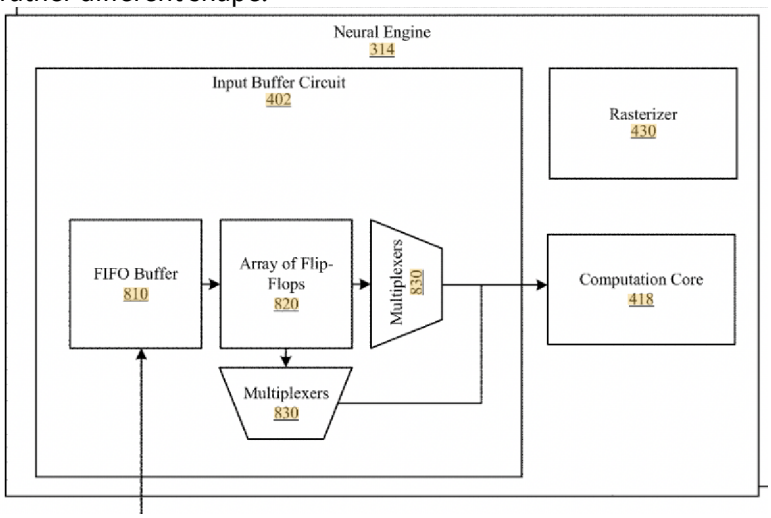
Suppose, for example, that we need to process a 17×17 array. If we have to cover this using power-of-two work units, we would need something like three 32×8 (= 256) work units covering a total of 32×24 to totally cover the 17×17 array, and only working on 38% of this 32×24, with the rest being zero padding.
The new design adds more smarts to the Smart L2, so that data can be reshaped (apparently now at a granularity of multiples of 2 or 4) to better match the work unit size (strange numbers) to powers of two. I suspect the Smart L2 mainly provides the data reshaping capabilities; and that the logic for how to do this is calculated by the ANE compiler and embedded in the ANE binary.
The reshaping moves the data within a power of two block (along with zero padding) and involves some fancy footwork (presumably co-ordinated with the Shifter and Rasterizer) to ensure that the "edge" work doesn't land up reading inappropriate values for the edge convolutions.



The essential idea is shown below. Our input is 24×10=240 elements. Naively we would have to split this (9A) into 10 rows of 32 (rather than 24) elements. We'd then have to send this to a Core as a first unit of 32×8 (=256) in size (with 8 of each row of 32 as to-be-discarded zero's), then a second unit of only 32×2 in size, so doing almost no work.

The alternative is to pack the data into two rows each of 128 in size. This fits into a single work unit, and only the last 8 elements are zero padding, so 120 of the 128 elements of each row are real data.

That's the goal of the patent. The bulk of the patent the describes elements of the logic in the Smart DMA and in the storage that feeds the work unit into the array of multiplier-adders, to actually implement this idea.
As you'd expect this is fairly technical; the one element that is easily described, and which may be new, is that the Input Buffer Circuit (the part that feeds signal to the array of multiplier-adders) is shown to now have an associated FIFO buffer, so that even as we are working through multiple convolutions acting on a particular work unit, we can transfer the next work unit from the Smart L2 to the Input Buffer Circuit, and immediately transition to convolutions on this next work unit, without a delay waiting for the signal to be transferred.
The arrays of multiplexers then help flow the data out of the Array of Flip Flops in an order that matches the desired convolution and array shape, even though what's stored in the Flip Flops is a rather different shape.

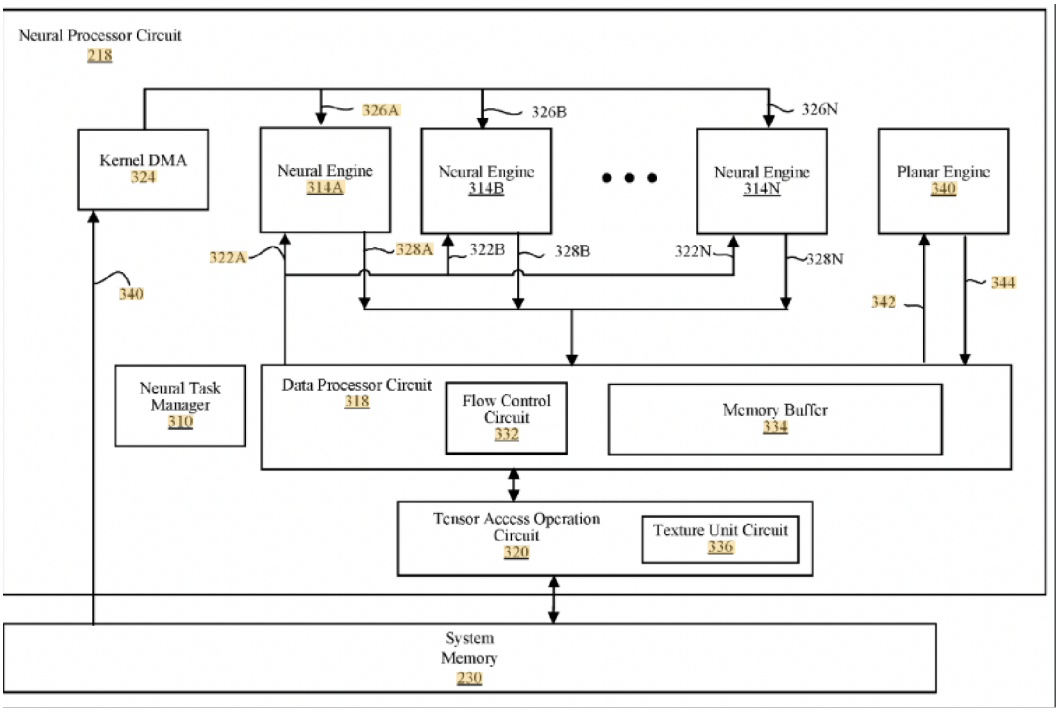


More sophisticated signal addressing and indexing

(2020) rank-5/6 tensor support

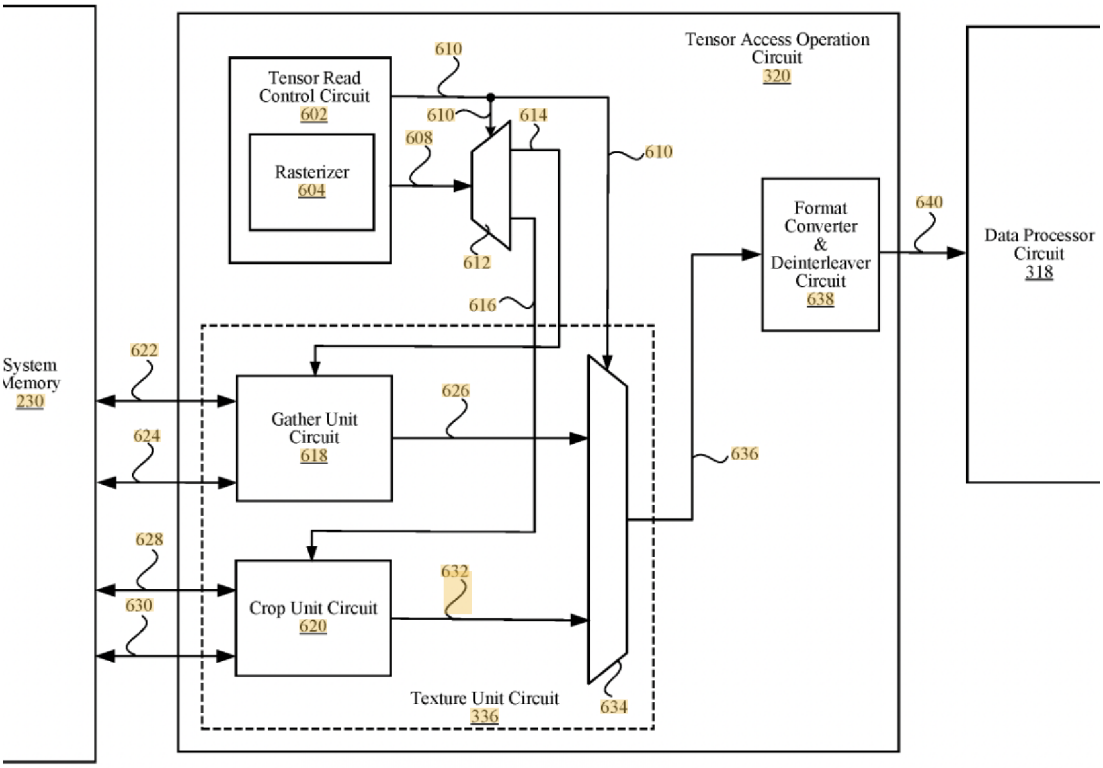
We've seen that in the design so far we have essentially rank-3 tensor support throughout the system (in looping through work units, through kernels, in executing DMA, in detecting when buffers are full and can be transferred between the Cores and Planar engine, and so on). The essence of (2020) https://patents.google.com/patent/US20220156575A1 *Multi-dimensional tensor support extension in neural network processor* is to extend this to rank-5 (or rank-6, depending on exactly how you count) support for the purposes of DMA (ie loading tiles into and out of the Smart L2).

The diagram looks familiar except for Tensor Access Operation Circuit 320, which governs DMA into and out of 318 (what I keep calling the "Smart L2"). [Also new is 336, which we'll get too soon...]



As you'd expect by now, this support for higher rank doesn't just mean that we can load rank-5 tensors, it means we can, during the load, reshape them (eg transpose the data between two indices) or extract slices (eg a rank-3 slice) from the full tensor.

But there's so much more going on here! Tensor Access Operation Circuit 320 provides all manner of new ways to move and rearrange data between system memory and the Smart L2, as shown in



Format Converter and Deinterleaver 638 seems simple enough (though we don't get details of just how much it can do. For example, can it quantize and dequantize *signal*, as opposed to the quantization and dequantization we've had since ANE 1.0 of kernel?)

You should really think of what's going on here as: remember when I talked about a new computing paradigm that treated rank-N arrays as a universal data structure, and how part of how that would evolve would be various schemes to pack/compress sparse versions of these arrays? Here's where we see that idea instantiated. The patent is unclear on just how much is provided (you can think of many different types of sparseness), but it seems to be pretty clear that what we definitely have is a way to construct "large" arrays on the fly from smaller arrays. Why is this useful? Well, think of what we said about LLMs. You start with a stream of words which you convert into a stream of tokens. Each token (say one word) then corresponds to a embedding vector (say an array of 768 FP16s). You'd far prefer to expand a token into an embedding vector as close to the ANE as possible, rather than storing a stream of (much larger!) embedding vectors in RAM.

So one thing this new Tensor Access unit can do is take an index array (eg an array of tokens) and for each token look up the value of that (possibly multidimensional) index in some "reference array" (like the array that maps tokens to embedding vectors). Generically we can now, instead of "direct" lookup of a rank-*N* array, we can perform indirect lookup, referencing a rank-*M* index into a rank-*N* "reference array".

(2020) "texture" lookup

OK, so we understand how the Tensor Read block gives us indirect memory access, and why that might be valuable (serving as one form of in-memory compression of the data stream we ultimately wish to manipulate). That covers one version of "tensors as a universal data structure requires transparent sparsity and compression". But another thing that PDF suggested was being able to index into a tensor using non-integer values, and getting interpolated results. That's what (2020) https://patents.google.com/patent/US20220138553A1 *Texture unit circuit in neural network processor* provides, as seen in the lower part (Texture Unit Circuit 336) of the above diagram.

In "indirect lookup" (called "gather") mode, the Texture Unit behaves as I have already discussed. Essentially Rasterizer 604 generates a stream of addresses that walk along some array of indices (call it the token array). These addresses go into the Gather Unit which makes a first lookup into the Token Array (based on the given address) to look up the token. It then makes a second lookup into the Reference Array, to lookup the embedding vector.

Even weirder is so-called Crop mode. Now imagine the Texture Unit as behaving like a GPU texture unit! The Rasterizer 604 generates a stream of "pixel lookup indices" which we want to look up as interpolated values within some array. So, like a texture unit, we need to crop the input values to

nearest integer indices, lookup up various values around these nearest integer indices, and interpolate the whole mess to an output value. And like a texture unit we have controls for what to do at edges (mirror, wrap, repeat the edge value, etc).

Why on earth would you want this? My first thought was for FaceID (and more generically image recognition). The standard vision neural networks expect input data of a particular size. You could rescale each image (and the FaceID "photo") to the desired size on the GPU, then transfer it to the ANE; but it's faster and more power efficient to do the job on the ANE. And this allows you to start doing things like run one neural net on a photo just to find "objects of interest", then extract and scale up each object of interest and submit that appropriately scaled up sub-photo to the recognition network to get an optimal recognition of each object of interest. My subjective experience is that FaceID on the iPhone 15 Pro (ie A17) is quite a bit faster and more forgiving of the phone being at a rotated or tilted angle than was my iPhone 13 Pro (ie A15). So I could believe that something like this is being done on the FaceID path (and then on similar paths, so not just image recognition but also things like finding areas of interest for translation when using the Translation app to, eg, translate Chinese). And of course, like we discussed for the kernels that could be flipped and rotated, something like this can be used to increase the pool of images being used to train a vision network.

nVidia at one point talked about how they kept JPEG decompression on their cards long after it seemed to no longer make sense, and said the primary use case was to decode images that were feeding vision recognition software, because otherwise that software was gated by how fast (much larger, uncompressed) images could be moved into the card over PCIe. The texture unit may fulfill the same sort of role; a way to condition images for recognition or training that removes GPU speed as the limiter on what would otherwise be much faster ANE performance.

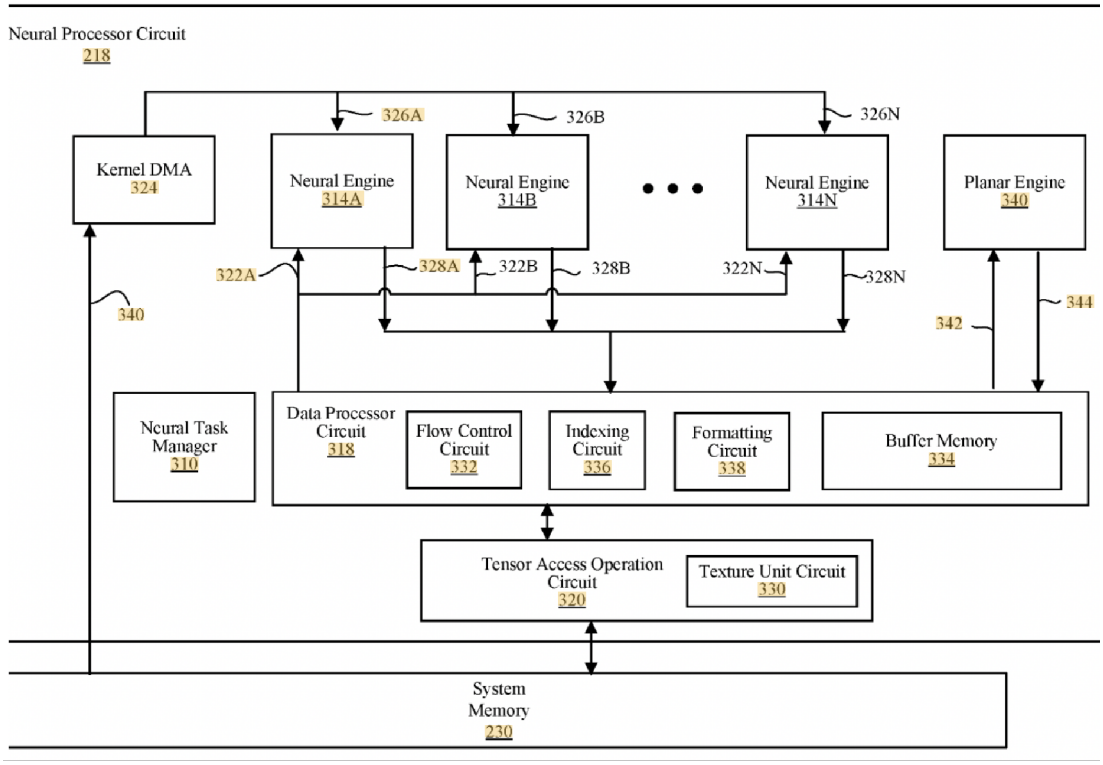
You could imagine other uses for this sort of thing. Perhaps skim through audio at high speed, until you detect "activity", then sample the active data at higher precision? Perhaps interpolate data within a 3D volume for Vision Pro?
And (most interesting of all) is any of this valuable once we get to the intermediate stages of an LLM, once we have transitioned from tokens to embedding vectors, and are implementing "semantics" and "grounding" as (interpolatable) points within some high (eg 768-D) dimensional space?

(2021) move indexing into the L2

The above functionality (indexing, and texturing) seems useful enough, but it is limited to data that is flowing through DMA, ie flowing from (or perhaps to?) system memory. (2021) https://patents.google.com/patent/US20230169316A1 *Indexing Operations In Neural Network Processor* moves a small part of the indexing functionality into the L2 itself, so that data or index arrays produced by either the Cores or the Planar Engine and stored in L2 buffers can be used for indirect access without having to route through system memory.

Once again, if we look at the diagram, we see the new features (within the L2) are Indexing Circuit

336 and Formatting Circuit 338.



The idea is as you would expect, we can use a index array (within Buffer 334) to construct a tensor on the fly from a reference tensor also stored in 334, while using Formatting 338 to align, shape, and pad the result as desired. Overall much the same as the earlier version, though somewhat simplified and stripped down.

As to why this might be useful, unfortunately we're rapidly moving past my scope of knowledge! I assume there was a real world reason for this, but I can't think of situations where this sort of indirect lookup might be useful, but might also fit into the few MB or so that are available in Buffer Memory 334.

Could you use this indexing stuff (in either the DMA form or the L2 form) to implement traditional sparse vectors and matrices? I *think so*, but not optimally. You'd some degree of memory compression (always nice) but you would not get compute unit compression. Maybe good enough for low levels of sparsity (say 50% or so) but not for extreme levels like 10% or less.

(2021) data retention in the Planar Engine

Finally we have (2021) https://patents.google.com/patent/US20230121448A1 *Reduction operation with retention in neural network processor*, which seems like an extension of (one element of) the above ideas.

You'll recall that the Planar Engine, along with everything else it can do, can perform reductions of its current patch of data, giving us the sum of elements in the patch, or the maximum, or whatever. A slight extension of the idea has always been that some of these values can be accumulated in the so-

called Line Buffer, but over a short time period, essentially to pass some data between the First and Second Filters. This patent allows that data to be retained for a longer period of time, and to be used more flexibly within the Planar Engine.

I think, ultimately, it's another version of moving data closer to where it will be reused. It seems like the original scheme was something like: data could be accumulated in the Line Buffer, moved (as a rank-1 tensor) to the Smart L2, then later reloaded for reuse against a different patch or different input tensor. But if we can retain the data in the Line Buffer for longer, we can avoid that trip to and then from the Smart L2.

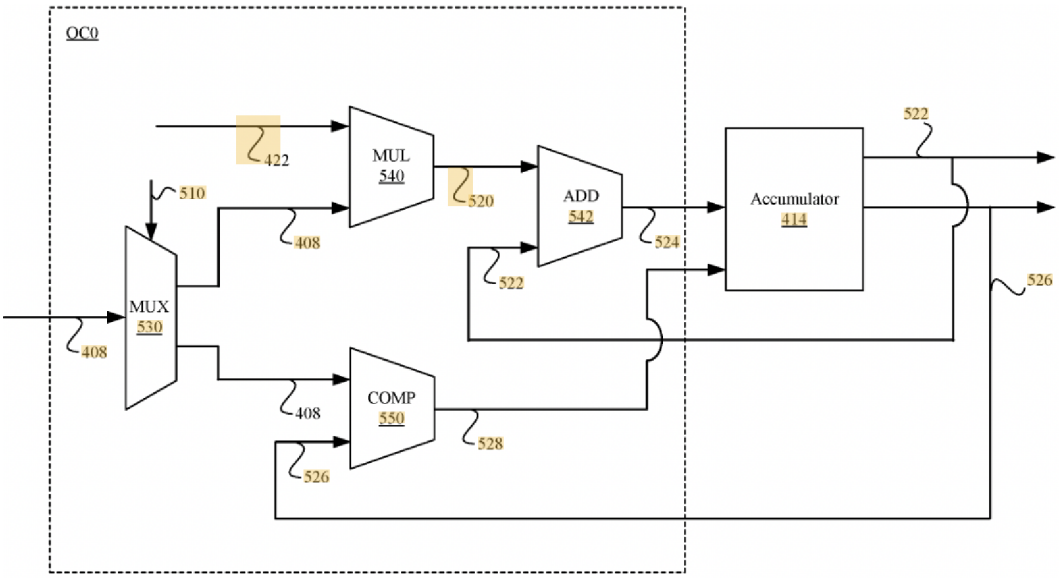
Sorting and Comparisons

(2021) simple sorting in one core

Now let's move into the Neural Core itself, with (2021) https://patents.google.com/patent/US20220222510A1 *Multi-operational modes of neural engine circuit*. The point of this patent is to allow (a set of) Cores to sort some data.

The first question is why do we even want to do this? It doesn't seem a very "neural" type of operation! I could find no good answer in the patent, or on the internet. The best I can come up with is that the end result of many neural tasks is a list of possibilities each with some degree of belief attached, so an image recognition task might spit out something like 90% dog, 7% horse and 3% cow. One probably wants to sort this list for presentation to the user (after sorting one can then decide to stick with the highest value, give all values above 1%, or whatever). One may also have multi-step tasks, something like a few ranked options come out of speech detection, and then go into some sort of image synthesis.
Sorting may also be important for LLM's. We'll discuss LLM's soon, and the sort of additional support they might require that's not naturally going to be present in a Vision unit.

Bottom line is that, for whatever reason, Apple felt it worth adding a sorting ability to the NN.
The change is not that large. The Multiply-Accumulator unit now looks like



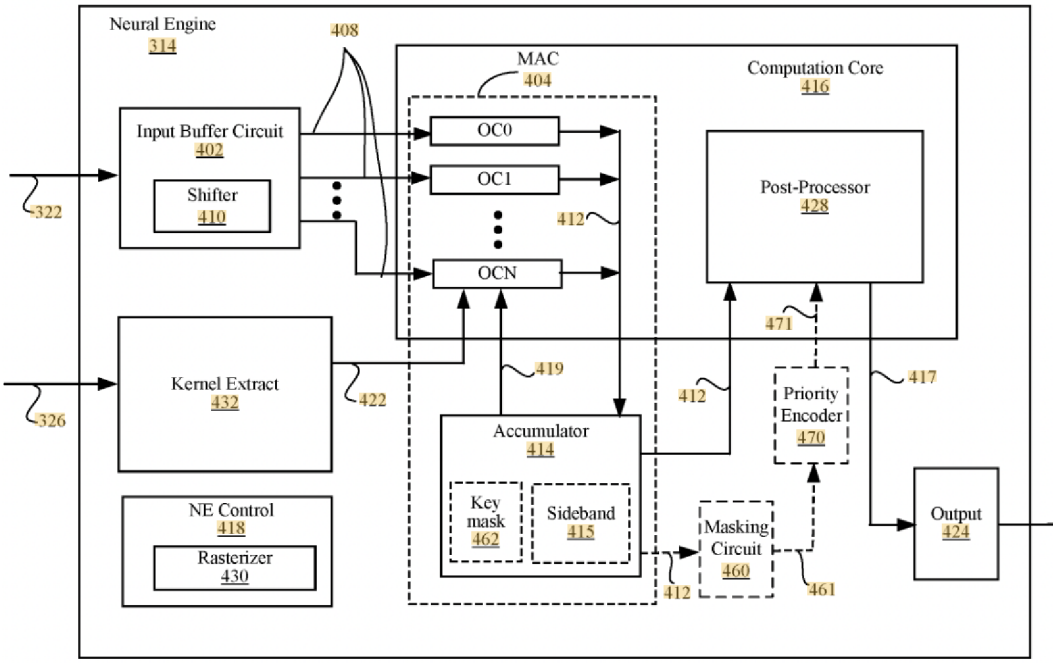
where the change is the addition of the comparator. We saw with the GPU that one can make a cheap comparator that tests >=< for both int and fp, without requiring the additional costs of subtracting two values from each other. The two values to be compared are a value supplied by the accumulator, and a value supplied as signal input; in this mode, the kernel input is ignored.

With this functionality you can do load the accumulator with 128 or 256 values, and then perform 128 or 256 compares. What does that give you? It does give you the basic elements necessary for a parallel sort, eg a bitonic sort. These sorts are $O(N \log^2 N)$ which is not as good as a "normal" sort of $O(N \log N)$, but the prefactor is so much better that they're overall a win.

The patent does not describe any details as to how you might stitch these 128 or 256 element sorts together, but we see more of the vision in two subsequent patents.

(2022) more sophisticated lexicographic sorting

First is a year later, (2022) https://patents.google.com/patent/US20230229902A1 *Key-based comparison in neural engine circuit*.
Building on this earlier framework, we create a fairly generic sort accelerator. We can now perform somewhat generic sorts (lexicographic, so byte by byte sorts of a string-like key) with the Accumulator storage extended to track progress through the lexicographical "string" comparison. In the diagram below, note the addition of Key Mask 462, Sideband 415, Masking Circuit 460, and Priority Encoder 470, which together (along with some work in Post-Processor) perform the byte-by-byte sorting.



The details are far from clear (especially since it's unclear what hardware is available, or even the exact problem to be solved!) but the bottom line appears to be that
- we have a parallel sorting engine that handles strings
- this can scale "fairly large", to at least 65536 records, each consisting of a a payload and a string-like key
- there is a mechanism to bail out of the sort before the end, eg to ask for essentially "give me the top 10 results" rather than "give me the entire collection of 256 sorted results".

It's unclear if this is all an important part of building an efficient LLM? Or if someone realized the ANE could be used as a generic sorting accelerator, unrelated to neural tasks... Although, as a sorting accelerator, the scheme described is somewhat limited (eg you cannot provide your own comparison function) for many purposes it may be good enough – all you need is *some* sort of a collection of keys, you don't especially care about the details of how they were sorted, just that they were sorted and now can be, eg, searched using a binary search.

(2021) element-wise comparison in the Planar Engine

Our second patent is (2021) https://patents.google.com/patent/US20230128047A1 *Binary comparison and reduction operations in neural network processor*. This gives us element-wise comparisons in the Planar Engine. We can pass in two planes, compare them in one of the filters (generating eg +1, 0, -1 outputs depending on the element-wise comparison) and then, optionally, reduce the resultant comparison values.
I don't think this is for sorting, more it's another form of delinearization. For example one way this might be useful is to binarize feature vectors. After calculating a set of feature vectors we have a set of real numbers; but we may only care about

- a few largest numbers and
- the fact that they are present, not the exact value.
ie we want to go from say a set of 256 FP values to a set of 256 booleans (0 or 1, or maybe 0, +1, -1) where the importance is in the index of where each non-zero boolean sits, ie ten or so of the 256 indices are ±1, the rest are zeros.

Why have I occasionally referred to *binarized* networks? Because of (2019) https://patents.google.com/-patent/US11669585B2 *Optimizing binary convolutional neural networks*. This is one of a few patents acquire by Apple when they purchased the company xnor.ai. The patent is very strange, reading more like a paper than a patent, describing various experiments conducted! But the bottom line is that the experiments suggested that many (not all, but many) of the layers of a vision neural net, could collapse both the layer-to-layer signal data and the weight (ie kernel) data to 1's or 0's, in which case the convolutions become simple bit manipulations, with substantial performance improvements.
This sounds interesting and worth following up! However it's unclear if it fits into the main flow of Apple's work. Certainly there do not so far seem to be these sorts of boolean operations available in either the Neural Cores or the Planar Engine; but to be fair it might take a few years first to figure out where this technology is useful (experimenting on CPUs and GPUs), and then figuring out how best to fit it into the existing ANE structure.

Stochastic rounding

As we've discussed to some extent, and as you will see when you read about neural networks in more detail, both in training and interference you generally want some randomness injected into the process. The formal network design may specify a particular type of randomness (eg Boltzmann), but if we can get away with a cheaper solution that's almost as good, why not do so? (2022) https://-patents.google.com/patent/US20230236799A1 *Stochastic rounding for neural processor circuit* describes one such cheaper solution.

Recall that once the multiply-accumulates for a given convolution in an ANE core have completed, the final value (which may have been accumulating as an FP32 or an INT32) is passed through a Post Processor step which, among other things, may perform a delinearization function (eg ReLU), and/or some sort of normalization, and/or rounding (eg from FP32 back to FP16). We can overload this rounding step by adding a random value (of variable size, and not necessarily limited to the size of the least significant bit of the input value; we can vary the amplitude of the randomness we inject), and thereby inject some degree of stochasticity into the overall process.

Intermediate memory reductions

A recent theme of ANE improvements has been ways to reduce the amount of working memory. The most naive way to execute a neural network proceeds layer by layer, meaning that we need to generate the full output of a particular layer before we move on to the next layer. Since intermediate layer

data can be large, this may mean a lot data (and thus time and energy) spent moving values to L2 and then SLC, and then back again. Can we avoid that?

A first scheme is (2021) https://patents.google.com/patent/US20220398440A1 *Circular buffering in neural network processor*. This adds circular buffering to the intermediate buffers already under Smart L2 flow control. So instead of, for example two large buffers required to ping-pong between some ANE Cores and the Planar Engine, so that ANE cores can fill one while the Planar Engine reads from the other, we now provide transparent hardware support for a single circular buffer that is filled at one end by the producer as it is simultaneously read from the other end by the consumer.

A second variant on this idea considers the situation where one convolution layer (along with eg a simple ReLU stage) feeds into a second convolution layer. Since convolutions are local (unlike eg a fully-connected layer) we know how much of the first convolution needs to be performed before we have generated a work item sized unit for the second convolution. We can thus structure the pair (or even more such layers) as a *streaming convolution* that flows the data from one layer to the next to the next, at each stage using the minimum necessary amount of intermediate storage, just enough to gather the data for the next layer. This is the focus of (2022) https://patents.google.com/patent/US20230368008A1 *Memory-efficient streaming convolutions in neural network processor*. The bulk of the patent discusses a formal memory layout for how this works (once again, the trickiest part, setting this all up, is probably handled by the ANE compiler). This is still done by flowing data through the L2, not by direct Core to Core transfer.
One interesting hardware aspect is that yet more smart addressing has been added to the L2, providing something like a generalization of the previous circular buffering, so that the producer and consumer convolutions can see the intermediate storage as a linear “tensor-addressed” memory space, even though it internally wraps around and operates as a circular buffer.
(In principle, with blocking, you should be able to also use the same idea for flowing fully connected matrix multiplies from one layer to the next, so limiting intermediate storage. How this plays out depends on Apple’s [unknown, so far, at least to me] strategy for handling LLMs...)

Next we have the related patent (2022) https://patents.google.com/patent/US20230394276A1 *Subtask storage for streaming convolutions in neural network processor*, which describes some necessary (but honestly not very interesting) details of how *subtask*s (which is apparently the term being used for convolutional layers that work together, like the streaming convolution we have described) are packaged in the ANE binary. “Executing” these layers is an exercise in setting up a whole lot of registers that described the DMA, the Smart L2 memory layout, the various shifters and rasterizers, etc; and this new functionality allowing two such convolutions to operate together, along with a fancy circular flow-control buffer linking the two, needs more details defined in the binary.

Rounding out this theme, we have (2022) https://patents.google.com/patent/US20240095541A1 *Compiling of tasks for streaming operations at neural processor,* which brings these strands together.

The ultimate goal of the patent is to be able to compile nets in a “streaming” rather than layer-by-layer fashion, which refers to what we’ve discussed above – detect layers that flow data from one bounded calculation to another calculation, along with situations where dependencies of some sort

do not allow this.

Once you have this scheme in place, the next step is to describe the "streaming" in the binary.

Finally, for this to be of maximal value, the scheme needs to save memory relative to the layer-by-layer alternative. This requires essentially tracking data dependencies from one layer to another. If we can say that a block of memory *b* within tensor *B*, about to be processed, is the result of calculations limited to block *a* of tensor *A*, then we can both schedule block *a*'s calculation to flow into block *b*'s calculation (memory locality) *and* once we return to calculating the next block of tensor *A*, we can reuse the memory of block *a*.

The technical side of the patent is the compiler techniques required for these analyses, first of layer-to-layer dependencies; and then of data-to-data dependencies and how those can translate into the reused of memory blocks within the limited ANE SRAM.

Reduce memory latency

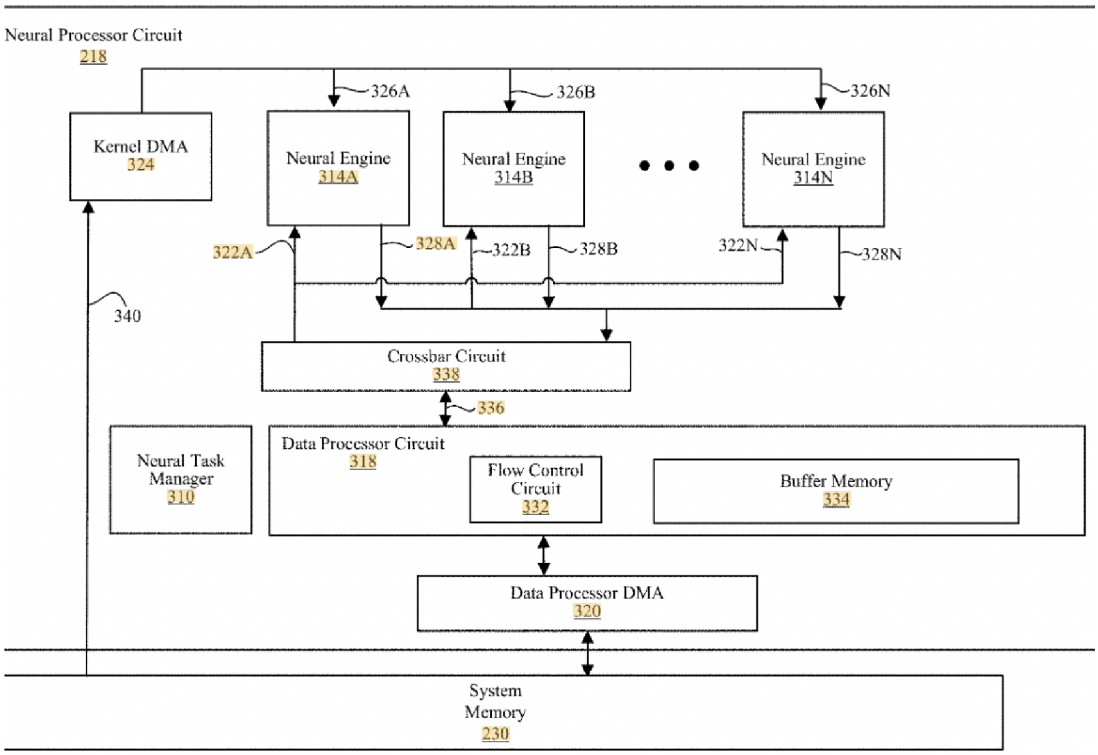
An obvious addition to the ANE is some sort of prefetch of data required by the next layer while we compute the current layer.
This is described in (2022) https://patents.google.com/patent/US20230289291A1 *Cache prefetch for neural processor circuit*.
Most of this is using ideas we have seen before.
Like Metal, the compiler knows the patterns of resource access, so it can provide in the binary prefetch hints, to be used or ignored by the hardware.
Like both Metal and the Display system, the prefetching may use a "sieve"-like scheme rather than a naive prefetch. In other words, rather than prefetching eg an entire data set (which won't fit into local storage) we prefetch say every even line, then load the odd lines while we are performing computation using the previous odd+even line pair, thus halving the bandwidth required during the second layer. Of course the details of what counts as a "line" (ie a sensible unit of prefetch size) are layer-specific and will be indicated by the compiler.

It seems like much of this work is done by the Neural Task Manager (which I am assuming is the ANE ARM companion core), co-ordinating the movement of prefetch data from system DRAM into the SLC. It's always a nice sign when we see that, because it suggests that, at least in principle, a good idea, like this data prefetching, could be implemented in firmware, and thus retrofitted to older Macs with a firmware update.

Memory crossbar

A somewhat technical patent is (2021) https://patents.google.com/patent/US20230135306A1 *Crossbar circuit for unaligned memory access in neural network processor* which describes the circuit that aligns data as it moves from the common L2 Buffer to the work unit storage within a particular ANE core. Structurally the most interesting implication (if we take the diagram seriously) is that only one line transfer per cycle is possible to the entire set of 16 ANE cores. This seems a little tight, so maybe there

are two of these units, each serving eight cores?
Technically, the basic issue is that we want to transfer a block of data that is fairly long and consecutive, say 64B, but has a random alignment in the L2 buffer. We need to handle that alignment (move from random alignment to “zero offset”, but we don’t need generic byte-level rearrangement, just something that moves the entire block to the correct alignment, at minimal energy cost. And that’s what the patent describes.



Power savings – and non volatile memory!

This one, (2020) https://patents.google.com/patent/US11385693B2 *Dynamic granular memory power gating for hardware accelerators*, is really strange! But it’s from people who have their name on a number of ANE patents, so it’s somewhat in the mainstream...

The primary idea is not surprising. We define the L2 storage buffer as a collection of independently powered SRAM blocks, and then we (which presumably means the compiler) can use our knowledge of the data flow and timing to save power. At the simplest level, we can sleep (use a minimal keep-alive voltage) SRAM blocks until we need their data; at a more aggressive level we can keep blocks unpowered until we are storing in them data that we care about, and then remove the power once that data is no longer of interest.

The first strangeness is this idea is followed by a discussion of what happens if multiple separate accelerators want to share use of this L2 buffer, and how a common power manager can handle the differing requests from the two or more accelerators. Is this a reference to the vector DSP possibility

that I describe below in the ANE 4.0 section?

The second strangeness is that the patent then veers off into discussing how to use a non-volatile storage array also available to the L2 buffer. The idea seems to be that this would be dense storage that might be expensive and slow to write, but once written could hold its (rarely modified) data, so most likely the weights of a neural net, over many cycles of the ANE being powered on and off. Storage that seems appropriate (sort of) for the task might be either MRAM or RRAM, both of which TSMC offers, and both of which are about 2x the density of SRAM. *But* TSMC, at least publicly, only offers these on somewhat lagging nodes, for example the roadmap says that N6 gets MRAM in, I think 2024.

So it's unclear how serious this idea is.

One option is TSMC and Apple have plans for MRAM on leading nodes sometime in the next few years. Another option is that Apple has plans for stacked chiplets, eg placing a fairly large non-volatile MRAM chiplet on top of all the A and M chips, and using the large available storage for various purposes (with neural weights as one plausible use case that's a very good fit to the technology).

More than one ANE?

Suppose you are the lucky owner of an M2 Ultra. Can you get extra value out of the second ANE block on your second Max SoC? Well, yes'ish – but, at least right now, in a way that's pretty pathetic.

This is described in (2021) https://patents.google.com/patent/US20220391677A1 *Kernel-level load balancing across neural engines*. The idea is that if you have one app making requests to the ANE **and** simultaneously a second app happens to make a second request that's best handled on the ANE, we route the second request to the second ANE unit. Which, OK, sure, better than nothing, but hardly impressive!

To some extent this seems like a software issue, and maybe with the work done on streaming (ie executing parts of a layer then feeding the values on to the next layer, rather than a full layer at a time) some effort will be put into splitting the blocks of streaming work into two halves that can execute on each ANE. Or consider the way most vision nets start by applying some large number of different convolutions to the input image, to detect edges; we could maybe split half of those convolutions to run on the second ANE.

What we definitely are not seeing yet is the same degree of thought as went into scheduling tasks across the Ultra GPU, though perhaps that will come?

There are similar other software issues that have not yet been addressed but may be over the next few years. For example there's seems to have been no attempt to split tasks across all compute (do some of a layer on the GPU, some on the ANE, some on AMX). Or the equivalent sort of splitting for the purposes of training where the task may be even easier given that most of the work is processing large

batches of independent material.

One thing we may see from the 2024 rapid change in the tech landscape is much more of this sort of extreme programming on the Linux and Windows side (it's the sort of problem that a certain type of programmer just can't resist!), which in turn will probably force Apple to do the same. I'm reminded of, for example, the initial work to allow the OS to switch dynamically between an iGPU and a dGPU; again somewhat inelegant in the details, but pushed through on the PC side, forcing Apple to achieve the same thing.

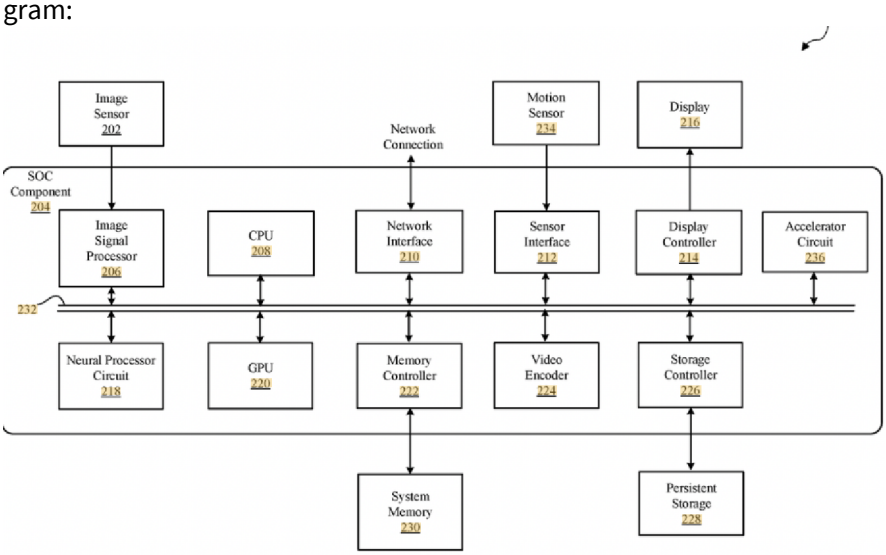
ANE 4.0?

The above describes what seems pretty certain about the ANE.
In addition to this, there are other Apple patents of a more uncertain nature. These generally arise when Apple buys a small company and picks up its patents (which may or may not be of any interest; Apple may have bought the engineers rather than the patents). The most ambiguous cases are when new hires (from one of these acquisitions) write up a patent that fits (kinda, sorta) into the mainline Apple stream, but the fit seems strained, and who knows quite what's happening!

addition of a vector DSP?

For example consider (2021) https://patents.google.com/patent/US11614937B1 *Accelerator circuit for mathematical operations with immediate values table*. The primary author is Liran Fishel, whose name is all over the first round of ANE patents. But the patent looks somewhat different from anything we've seen before.

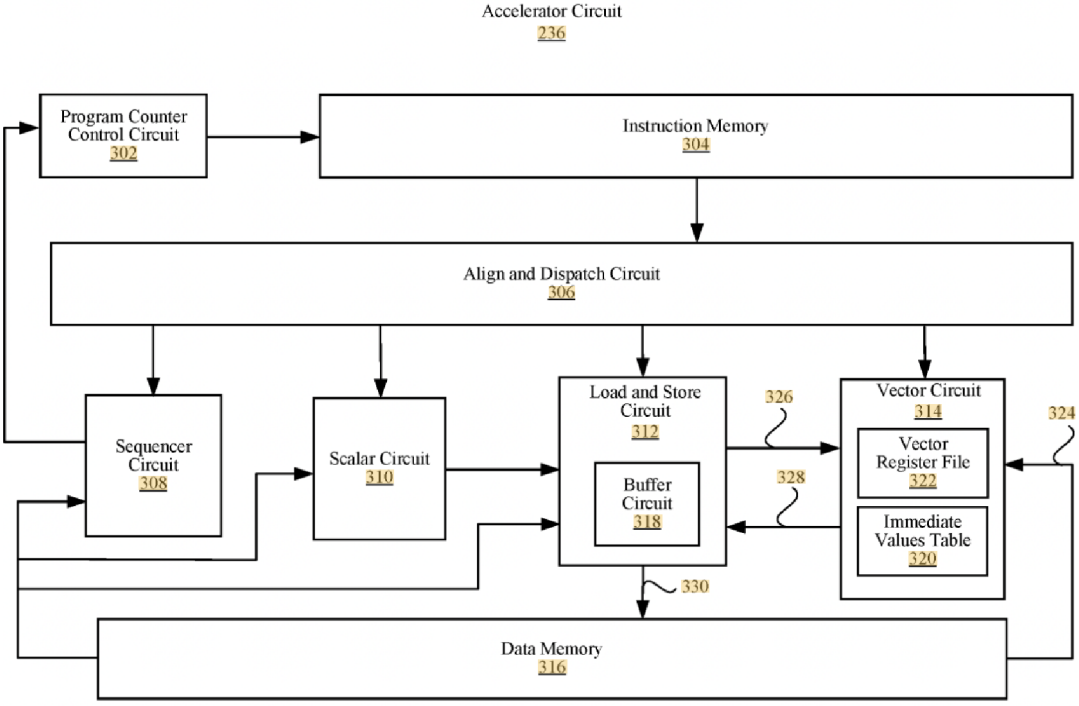
We start with an overview diagram:



This is something we've seen dozens of times, so surprises, except for the item on the far right, the

“Accelerator Circuit” 236.

This appears to be something like a vector-based (ie wide SIMD) DSP.



Meanwhile the patent talks about this in a context on neural networks and how they need dedicated hardware, blah blah. The patent also talks about this being “merged into” another IP block, like the ANE. One possible way of handling this is imagine the ANE starting to look something like a CPU cluster, with common L2 storage, but with a few someone separate computation engines (the ARRM companion core, the Neural Cores, the Planar Engine, and now this Vector DSP) all grouped together. This would make the Vector DSP usable by other hardware (I guess...) but still close to the ANE, if that’s expected to be the primary client.

We can see that overall this looks more or less standard for a DSP. We have what’s essentially an L1I cache, an L1D cache, a scalar engine for handling things like loop/branch/flow control, and most of the work being done in a SIMD engine. The instructions are the usual DSP VLIW design, being split and sent to the appropriate targets by Align and Dispatch 306.

What’s the point? Well at various stages in this discussion of the ANE, I’ve suggested various limitations of the hardware. For example what looks like a clumsy and limited ability to make any sorts of branching; or (more generalized) no sort of “scalar” processor to handle control plane (as opposed to data plane) decisions; or a need for hardware that can handle various sorts of specialized (and varying) functionality, like drawing values from a defined distribution, or calculating special functions (like exp). This vector DSP looks like a fairly good match to those sorts of problems. If we look at the competition (of a sort...) someone like Ceva provides a Vector DSP for use in digital communications (eg 5G

Baseband) that provides 128MACs per cycle and 256B wide memory support. That seems around what might be a good fit to ANE.

One more item suggested by the patent is FP32 (maybe even FP64) support. That seems out of line with ANE, but may be part of an Apple desire to keep the part somewhat general purpose? Or perhaps the idea is that certain "lighter-weight" FP32 neural tasks can be handled here more cheaply than moving them to the GPU or to AMX?

The specific patent is about a constant "ROM" provided to the DSP, called the Immediate Values Table. FPUs have had such ROMs forever, holding various constants like e, π, $\sqrt{2}$, etc. What seems to be new here is that the "ROM" is in fact programmable so that the constants can be updated depending on the function to be calculated. There's probably also some clever wiring in the actual logic design so that each constant value can be stored once, but loaded and broadcast to a vector register that's 32, or 128, or whatever elements wide.

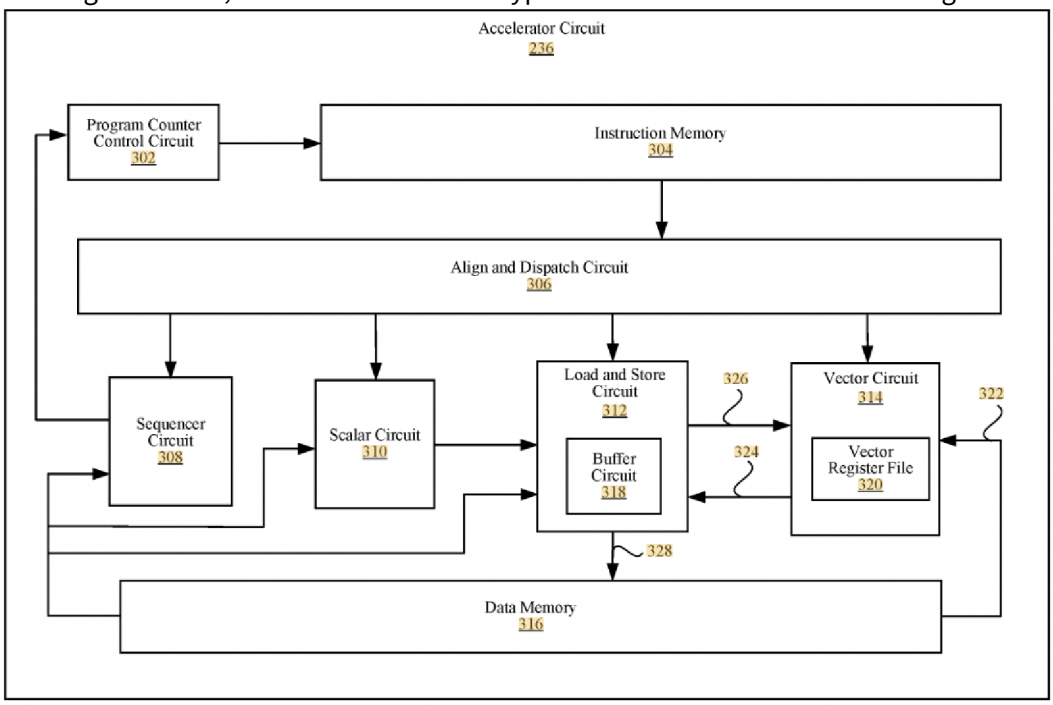
So that's rather neat! Who knows when Apple will tell us about it. Maybe it's already shipped?
As followups we have the rather technical patent (2022) https://patents.google.com/patent/US20230281106A1 *Debugging of accelerator circuit for mathematical operations using packet limit breakpoint*, describing one way of halting the DSP (so that registers can be read and the whole thing somehow debugged) by setting an instruction limit counter, so that after the device has executed that many instructions it halts.

And (2022) https://patents.google.com/patent/US20230267168A1 *Vector circuit with scalar operations in accelerator circuit for mathematical operations*. The novel thing here seems to be that there's some flexibility in what elements of a "vector" we use in a SIMD operation. The most obvious case is that we operate with (add or whatever) a full SIMD vector against a full SIMD vector. But we can also specify a single element of the SIMD register to be executed against a single element (like ARM can do, but now we can specify any two element, not just the lowest element of the two source registers). We can also specify certain variants like an *n*-element subset of each register, or an *n*-element subset from one register executing against a single element (expanded to *n* values) of a second register.

Why would you want this? The obvious answer is that it allows you to perform matrix multiples (either matrix-matrix, or matrix-vector).
We have seen how you can perform a matrix multiply as a sequence of outer product operations (AMX does this using an 8×8 outer product for doubles). Alternative sizes for the outer product are possible. In particular, an option that seems weird, but makes sense, is to have one "side" of the outer product be a long vector (eg say 128 or 256 elements) and the other be a short vector (8, 4, 2, even just one element). If you make the one element choice, then essentially your matrix multiplication consists of a carefully ordered loop of multiplying one long array against a scalar and accumulating to another long array, over and over through a sequence of scalars, then updating the input long array and repeating.

This can be (and **is**) done with something like NEON. But you can make it even more efficient if you can load all the scalars into a second long array, and then just select the scalar of interest each cycle. Alternatives are possible (reload the successive scalars each time, or shift the array right by one element) but both do more work than is really necessary.
Also this scheme is fairly easy to generalize if you want to make it faster. Just provide twice as many FAMC units, and flip one in the instruction to indicate that successive pairs of the "scalar" input vector should be used, to accumulate to two successive accumulator vectors. (And of course, if you really want to add hardware, you can extend to four sets of FMAC units or even more...)
It should be clear the similarities of this scheme with the Neural Cores, and in the same way it's trivial to at least avoid the energy costs of zero multiplies, and with a little extra hardware we may also be able to skip over a few successive zero scalars in each cycle, so also avoid the performance costs of zeros at least on the "scalar" one of of the two matrices being multiplied.

Reading both these patents, and with the apparent evidence (as of May 9 2024) that the M4 has add support for SME (presumably in the AMX block) it's natural to ask if these patents are in fact more relevant to SME/SVE than to a vector DSP. I just don't know, but they don't *look* like they're either an SVE augmentation, or relevant to an AMX-type block. Look at the structural diagram:



Just doesn't smell like AMX or the existing NEON units, does it?

BTW, fun side note.
Apple are, of course, currently engaged in a massive slog to build a modem, and who knows how that will turn out. There is a huge collection of relevant patents (constantly updated) so work is ongoing. Someone needs to review it because I certainly won't! However I do occasionally skim it quickly. What's interesting is that that much of the work involves a lot of the same sort of functionality that we have seen in the ANE – DSPs, vector and matrix math, use of the lowest precision you can get away

with to save power/area, parallel sorting, etc etc.
Right now it seems like the ANE team has zero interaction with the modem team, but perhaps (especially after the first modem actually ships) that might change? At which point there might be a very fruitful five years or so as each team swaps ideas and circuits with the other team...

Another interesting side note is that Vision is, of course, closely allied to many aspects of ML, and we saw that the ANE split off functionality and a design that was originally present in the ISP. The ISP remains the locus of much vision processing.
There are many elements to this vision processing and the capabilities of the ISP, but as just one example let's note (2022) https://patents.google.com/patent/US20220270208A1 *Blended neural network for super-resolution image processing*. As you can tell from the name, the goal is image upsampling, done through a combination of standard image processing techniques, also applying a neural network to the image (image passed to ANE via memory), and then appropriate patch by patch blending of these two images. This sounds good, and is a nice example of two IP blocks working together.

What is, however, not clear to me in all this is how flexible the scheme is. To what inputs can be it be applied? Presumably it can be applied to photos and video (eg to improve digital zoom). But can it be applied to user-supplied images or video (as in an API I can use to upscale a video file)? Can it be applied to what's flowing through a Display block (eg video game content, or Apple TV content) to provide frame-by-frame higher quality upscaling? You'd hope the above are true, but I fear that, at least right now, the scheme was designed to improve digital zoom, without thought as to how it might provide value in other scenarios. I fear (without good justification except the limited pattern I see in the patents!) that, unlike the ANE, the ISP is still tightly coupled to the camera data flow, and can't be plugged into modifying a stream of data from eg the GPU or the Media Decoder.

never to be used? or part of Apple Watch or Airpods?

An example of less obvious relevance to Apple's product line is (2019) https://patents.google.com/patent/US11410014B2 *Customizable chip for AI applications*, apparently picked up as part of the xnor.ai acquisition. This appears to be describing a very low power AI chip (with power provided, or augmented, by a solar collector or energy scavenging system).

the weird (PiM)

I've mentioned PiM (processing in memory) in other contexts. So far it's one of those ideas that sounds plausible, but also always seems not quite ready for prime time, always slightly behind what's achievable by other means.
Consider, for example (2021) https://patents.google.com/patent/US11694733B2 *Acceleration of in-memory-compute arrays*.
The idea is to perform the same sort of task as the ANE convolution engines, but by having
- a DAC converts to the multiplier inputs to a voltage
- an analog circuit multiplies the two voltages
- a wired-or of these analog multipliers effectively adds the results

- which are then sent through an ADC to give us a digital value
And all this is supposedly dense enough that it can be placed near to/in/on some sort of dense memory.
Does it make any sense? Will it ever ship? Your guess is as good as mine!

software/algorithms

There are a number of patents related to the software side of neural nets; some relevant to a neural compiler (eg ways to save memory, ways to fuse layers, ways to decide on the optimal target [ANE? GPU? CPU/AMX?] for a layer); some having to do with algorithms (eg various tricks to speed up training, or reduce its memory footprint).
But I don't really know enough to comment on these, so probably best to leave them for someone else to investigate.

what's missing?

When you look at the ANE hardware, it looks like a pretty good fit to current neural nets; and obviously we want to avoid having to move to the GPU or CPU as much as possible. So is there anything obvious that's missing?

It will be interesting to see if the vector DSP is real, and how it is used. It solves at least one set of limitations I've seen.

Another missing item I saw that might be useful, and might best be done in specialized hardware, is some sort of histogram facility.
There is a histogram facility in the ISP, and these histograms are sent as data to the ANE during camera processing. But there are other uses for histograms in neural net training (and ?perhaps? inference). For example during training, especially if you are optimizing a model for sparsity, you may want to use a "shifted ReLU". The way this works is that rather than a simple ReLu, which cuts everything off that's negative, and which may zero out 50% or more or less of your weights, depending on their distribution about zero, you run a pass to generate a histogram of your weights. From that histogram you identify the value below which 95% of your mass sits (or whatever sparseness level you want), subtract a constant offset to move that value to 0, then run a standard ReLU pass.

Part of what we are doing at early stages in a network is trying to extract "features" from the signal, like edges. Convolutions are a heavyweight way of extracting such features, whether from audio or images/video. Transforms are a more sophisticated way of doing the same thing. The traditional math transforms, like Fourier, are clearly suboptimal because localization is an important part of the feature extraction process. But what about localized transforms? That is, instead of doing the current per-patch convolutions, we perform one of DCT, DFT, Hadamard/Wavelet, or similar? These can be done at lower energy, especially in specialized hardware, and (with appropriate modifications to the subse-

quent layers of the networks) might be just as good?
This seems like it falls into the same bucket as the point I keep stressing, just how little optimization and engineering has been done of the various current networks, to move them from something that works into something that works at optimal area and energy.

Other reading

Having read the above, you may now want to look through (2017) https://eems.mit.edu/wp-content/uploads/2017/06/ISCA-2017-Hardware-Architectures-for-DNN-Tutorial.pdf *Hardware Architectures for Deep Neural Networks*. It's a nice quick overview of various alternative ways of doing things, and you should now see many familiar themes including
- ways to reduce the memory size of weights (quantization, pruning and omitting zeros, binarization)
- ways to reduce the costs of generating activations (various precision math, ways to exploit narrow math, eg INT8, especially by normalizing across layers – there's a reason the Planar engine seems so obsessed with normalization operations!)
- PiM-type hardware

The hardware they describe that seems most relevant to the ANE neural cores is mostly informed by Eyeriss, an nVidia/MIT experiment at around that time. This was unsuccessful in that it did not lead to any substantial changes to nVidia's game plan. Bill Dally (of nVidia) says that its primary mistake was that it was not *output-stationary*, meaning that it did not collect the activations, as they were being accumulated cycle after cycle, in static registers. That sounds good, but other articles suggest the optimal lowest power is a weigh-stationary architecture. What Apple is doing seems to be both weight and output stationary! So I don't understand much of the language people are using (and every year it seems to change anyway, as the leading edge of what's required in ML hardware keeps changing).

I cannot make heads or tales of most of this part of the tutorial PDF, and I don't care enough to look into details, but I think it's correct to say that the two things Apple got right compared to Eyeriss are
- they don't impose a spatial structure on their FMA units, they just have a flat set of 256 of these, with the relevant "spatial structuring" handled by the hardware loop feeding data from the work unit storage into the FMA's.
- intermediate convolution values are accumulated in a fixed register attached to each FMA, so output data movement is minimal during the calculation.
By luck, or by clever analysis and earlier experience, Apple seems to created what seems to be generally agreed to be the lowest power implementation of hardware for CNNs.

Something the tutorial doesn't state explicitly, but which is worth remembering, is why binarization might be interesting. It is known that for both DRAM and flash, you can play "analog" games with the device via the memory/flash controller (eg violate timing spec, or create a charge path between two rows that shouldn't ever be connected) to perform various bitwise operations between the rows, so

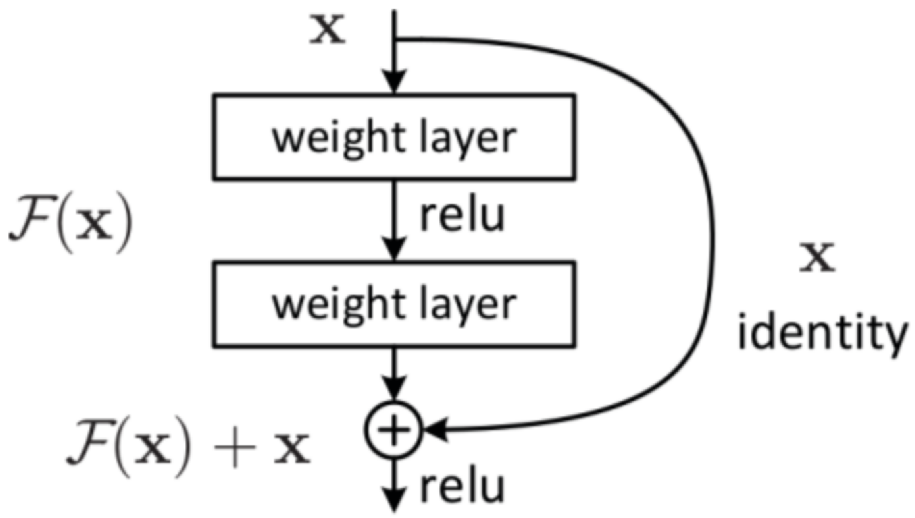
that eg you can perform a bitwise and or a bitwise xnor between the rows. Superficially, this suggests that you could then maybe “execute” the neural network directly in DRAM or flash, and avoid the energy and latency costs of moving data.
Of course it’s not that easy! For example binarization also requires the equivalent of a popcount in various places, to count the number of 1s; along with other operations like Pooling and ReLU. Even so, there’s the possibility, perhaps, of creating a combination of additional DRAM functionality and some logic very near [ie on] the DRAM to capture this benefit...
And, of course, some of this will likely not happen until LLM progress slows down! No-one wants to invest a fortune in a very clever, very specialized LLM memory that, oh dear, only runs last year’s model, while all the hot new functionality is only available to this year’s LLM. (One could argue that the original ANE to some extent suffered from this, being too optimized for Vision, and Apple has been lucky that the
on-going relevance of Vision has give them a few years to pivot the ANE a little more towards better Language functionality; probably over the next year or so we’ll see how effectively they used that respite.)

A second interesting paper is (2023) https://arxiv.org/pdf/2206.04040 *MobileOne: An Improved One millisecond Mobile Backbone*. This is a paper from Apple discussing how they put together and modified a wide variety of ideas from a wide variety of pre-existing mobile vision networks to create an Apple-optimized (and presumably ANE-optimized, though they never say this) vision network that can be adapted for various tasks. I don’t know enough about this space to understand about 80% of it, but a few themes shone through.

The reason they like ReLU is obvious enough,

and the points they made about the low correlation between performance (ie network latency) and both number of parameters and FLOPS required are important to remember in a world where you are constantly told that the FLOPS of NPU hardware, and the number of weights of a network, are the two most important numbers.

However the most important background point in the paper, always present, never stated explicitly, is that Apple really really hates branching networks. Remember that, in the context of neural networks, a branching network is something like



The above is a particular branching variant called a *skip connection*, where we send the data down to paths (one of which, for a skip connection, is the identity, ie we don't transform the data), and then we merge the transformed data of the two or more branches in some simple way, like an element-wise add.

Now, in principle, there's no obvious reason why this should behave badly on ANE. The problem is that certainly the current CoreML compiler, and perhaps the hardware, enforce execution layer by layer. Recall that we saw a few places where the compiler, for example, flattened neural graph to a linear execution of layers, adding dependency hints between the layers to ensure ordering even if one layer executed on the Neural Cores while another executed on the Planar Engine.

If we take this layer-by-layer literally then we have a problem. Obviously the optimal way to execute something like a skip connection is to execute a workunit (say something 256 or so elements in size) of the input *x* through the two weight layers, and then add the skipped *x* in the Planar Engine. But that requires primitives in the binary and in the hardware to co-ordinate the data flow. It's much easier (and certainly the way the early ANE did things) to run the full first layer on *x*, then the full second layer, then add in *x*. And this will require a lot more traffic, hopefully only to SLC, but maybe even out to DRAM, to hold the original value of *x* till it can be added back in. Hence latency...

If you read various Apple papers, even the most recent ones (which have publication dates of late 2023, and seem to refer to work done three years earlier, in about the A14/M1 era) they are constantly trying to replace skip connections with an alternative layer design that does the same job, but without a branch.

Now, some of the most recent patents I described, like *Memory-efficient streaming convolutions in neural network processor*, seem to suggest that the newest ANE hardware is capable of operating at this work-unit by work-unit granularity, rather than only at a layer by layer granularity. The desire to support things like skip connections (and of course the usual on-going attempt to reduce power by

using data as much as possible locally before being forced to move it back to SLC) may have driven this. If I'm correct, then over the next two or three years we may see a different set of emphases in Apple papers, with fewer workarounds attempting to avoid branching.

Another interesting thing about this tutorial is that, of course, as was the fashion of the time, it's all about vision. With a feeling of being somewhat irrelevant these days when LLMs are all the rage.

The final thing that's very interesting about this paper is that it's part of a series of papers, maybe 10 or so, see the list on this page,
https://github.com/apple/corenet?tab=readme-ov-file
that all seem to be about engineering optimization. I've mentioned on multiple occasions that so much in neural nets feels hacked together, not yet investigated and optimized; and in these papers Apple engages mostly in that sort of optimization. In various places they investigate things like the effects of different activation functions, simplifying nets, different training regimes (why keep training on images that have already proved easy to recognize?) etc etc. None of the papers is exactly a systematic list of "here are all the things we optimized and what we learned", but each paper has two or three such elements in it.

Final thoughts

At this point we know enough about the ANE, neural networks in general, and Apple's neural networks in particular, to be somewhat more intelligent and skeptical about what we read regarding this space, particularly discussion of benchmarks and TOPs.

CPU's have been around long enough and are understood well enough that we kinda know what to expect from a CPU benchmark (though even there a coder can write code that will, or will not, be well mapped by the compiler onto SIMD...)

GPU's take us into a world where coders are often much less aware of best practices for a given family of devices, so that even things like choice of algorithm or tuning (how many registers to use? how large threadgroups? ...) can have a large effect; and benchmarks are somewhat less valuable.

And with NPUs it's the wild west. The people writing these benchmarks mostly have no clue what goes on inside the NPU, have no clue what flexibility is available in terms of tweaking the net to get same results but better performance on hardware X, and I'm not even convinced they (or anyone outside a few large companies!) even know the appropriate balance of functionality to be testing.
Should they be testing mostly fully-connected layers or convolution layers? If convolutions, small (3x3) or larger (17x17)? What activation function should be used, good old ReLU, or various fancy alternatives? Testing with or without quantization? Straight line nets, or branched nets, or conditional nets (like mixture of experts)? etc etc

I suspect the whole thinking behind the TOPs claims and benchmarking is misguided, not engineering and not even marketing; more zombie marketing or cargo culting – enacting out rituals with no idea of

the purpose of the rituals. Let me explain.

When QuickTime started, it wasn't clear how things would play out and for the first few years the most visible face of QuickTime (not technically the most interesting, but publicly the most visible) was that you could plug in different codecs for audio, video, and images. I expected this to go on forever, with the collection of codecs growing indefinitely.
But after a few years smarter people than me within Apple realized that this was sub-optimal, that it made more sense, now that the world of codecs had settled down somewhat, to choose a few blessed codecs, optimize those as much as possible, and ignore the rest. So we went from a smorgasbord to one blessed codec: Sorenson (for a few years), then MPEG-4 for a few years, and then the world of today (h.264 then h.265, on optimized hardware for both encode and decode).

My point is that while the rest of the world is still excited about plug-in neural nets (as in I download a random net, equivalent of a random codec) and just start running it on my hardware, and likewise for developers, I suspect Apple is looking to a future where there are a few blessed nets (probably different on each platform) that are the workhorses for that platform, and which will be the primary targets of HW and SW optimization. Maybe MobileOne for vision, OpenElm for language on Apple, and the two equivalents on Android (from Google) and Windows (from Microsoft).
Just like you can run Dirac (or whatever the darling open source codec is this year) on your Apple hardware – but the experience will suck compared to just doing what Apple tells you and using h.265, so I suspect what will matter going forward is Apple's vision and language networks, not the performance of some random network. You will hook into Apple's optimized networks using Vision and Language APIs, with the ability to tweak things for various scenarios (eg adding vocabulary),Apple will focus their attention primarily on improving performance of those networks.
You can bring your own network and if it's for something minor (like classifying exercise) it will be fine; but if it's something major (like language) it will clearly suck compared to using the Apple built-in,not because Apple doesn't provide enough "TOPS" but because a developer targeting five different NPU architectures (none of which are well understood) is never going to do as well as Apple (or MS or Google) targeting a single well-understood architecture and engaged in software/hardware/training co-design and co-optimization year after year.

So I SUSPECT that this benchmark/TOPS talk is just nonsense. It's like boasting that you have hardware that runs Dirac way more efficiently than an iPad – perhaps true, but also utterly irrelevant.
The job to be done is not "run Dirac", it's "provide a video codec and its ecosystem".
Likewise for the average person, the ML job to be done is not "run <any random AI network> efficiently", it is "provide Vision functionality and Language functionality". The Apple built-in will do that.

It took a few years for the PC ecosystem to pick up this particular change in codecs (inevitable given the way the PC world is about fragmented hardware) and it will be the same for ML.
I expect three years from now QC, Intel and AMD, will be still be making these irrelevant boasts about benchmarks and TOPs (though not in ten years), while this simply will not matter on the Apple side; by

then the built-in Vision and Language APIs will be well entrenched, sane developers will not be adding their own large nets, and no-one in the Apple world will care how <random AI benchmark> performs on ANE; the only benchmark that matters is how Apple's Vision and Language API's perform.

Misc stuff I'll probably never get to, someone else should followup

Camera/ISP details I need to look into

HoG capture allows, eg, indicating camera is not aligned
temporal allows eg autofocus and long captures
other features allow matching points between images, allowing for image stabilization

Three stages – fixed hardware, configurable, programmable
Explain might be something like median filter or removing outliers, (and pooling can be eg local normalization)

2015-08-31 https://patents.google.com/patent/US10269095B2 Dynamically determining filtering strength for noise filtering in image processing

Multiple patents showing structure of the ISP, but without the vision processing part (and so not yet programmable)

2016-06-30 https://patents.google.com/patent/US9858636B1 Configurable convolution engine

Now we get a VPU and configurability, not real programmability.

https://patents.google.com/patent/US9992467B2 Parallel computer vision and image scaling architecture

gives the aspiration but few details (conceptually the wrapper patent, but mostly empty)

Important features now include

- array of multipliers feeding into array of adders
- 8x8 int multipliers generating 16b? integers. HDR handled as high and low planes of 8b data, separately filtered, then low plane shifted by 8 and added to high plane as the very last stage
- programmable pattern pulling data out of the input stream (eg to pool interleaved RGB [Y calc] or to separately filter R, G and B; to move vertically and horizontally sideways
- N-element filter is calculated over N -cycles. filter taps accumulate in time, output pixels are multiplexed in space

- filter values are broadcast(?) to all multipliers and latched in the multiplier

- “multiplier” unit can do other operations like compare or subtract? [or feeds the so-called compressor which does add/compare/min/max?]

- can latch two values from the input signal to perform same-element calculations like subtracting one image from another?

2017-04-27 https://patents.google.com/patent/US10176551B2 Configurable convolution engine for interleaved channel data

Boost performance by providing two engines (which can either process top and bottom half separately, or first does filter A and feeds into second doing filter B). Each engine in turn has two halves which each process half the interleaved data.

2019-03-19 https://patents.google.com/patent/US20200302582A1 Image fusion architecture (multiple lenses, fused to single image

2020-04-13 https://patents.google.com/patent/US11488285B2 Content based image processing removes the vision section from the standard ISP diagram. see also fig 5

2020-08-06 https://patents.google.com/patent/US20220044372A1 Image Fusion Architecture with Multimode Operations
2021-02-10 https://patents.google.com/patent/US11836889B2 Dual-mode image fusion architecture
2021-11-03 https://patents.google.com/patent/US20230138779A1 Linear transform of undistorted image for fusion
2022-03-11 https://patents.google.com/patent/US20230289923A1 Machine learning based noise reduction circuit

uses “ML” for noise reduction - but dedicated circuits, not ANE

then we wander off into Vision Pro camera patents...

There are other “Apple Neural Engines” of unknown use and structure. eg
2019-09-25 https://patents.google.com/patent/US11175898B2 Compiling code for a machine learning model for execution on a specialized processor

describes compiling for a neural engine with low memory capacity on AirPods! What is this used for? No idea.

2019-02-11 https://patents.google.com/patent/US11410014B2 Customizable chip for AI applications

describes something like ANE implemented on an FPGA, and reconfigurable for different tasks. Strong emphasis on low power (powered by energy harvesting or solar) and the sample net they give looks vision based. Maybe security camera? Maybe for the Apple car to provide Sentry mode w/o using much power?