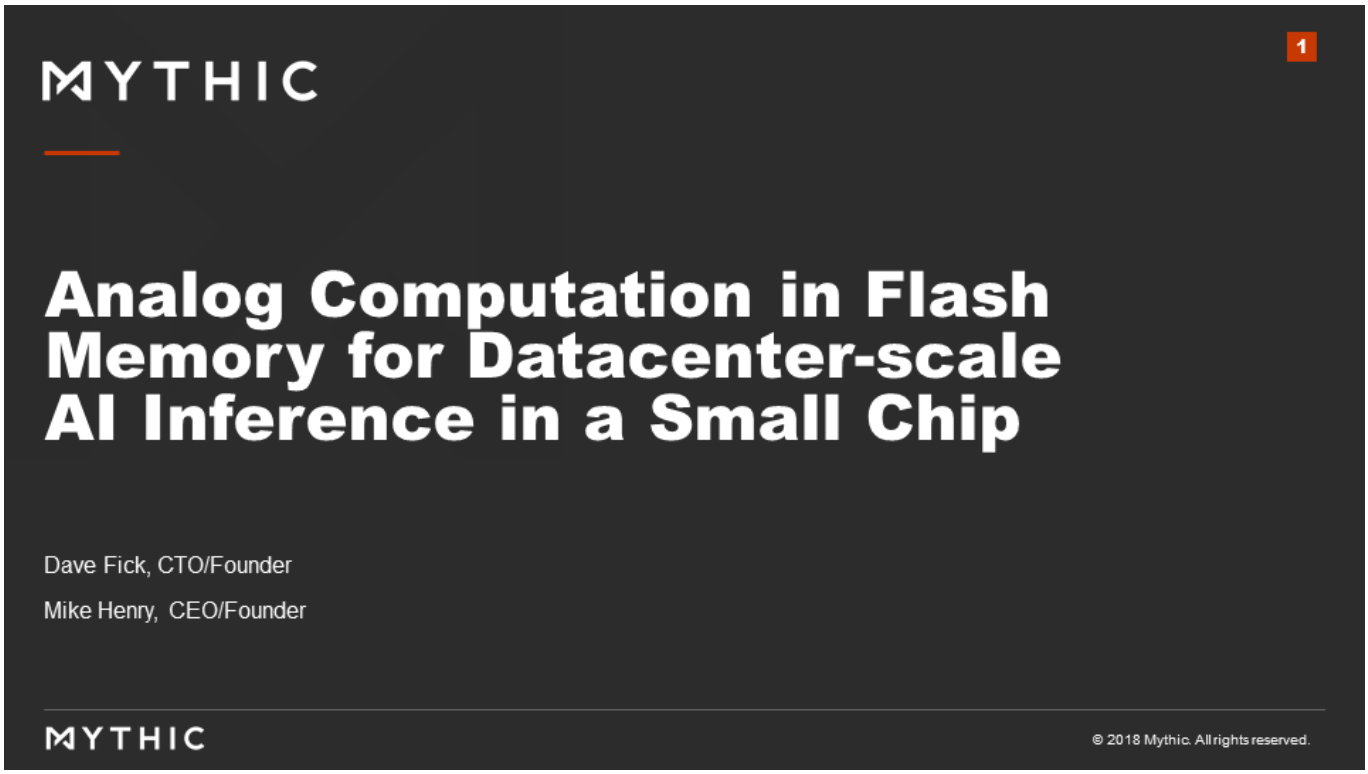
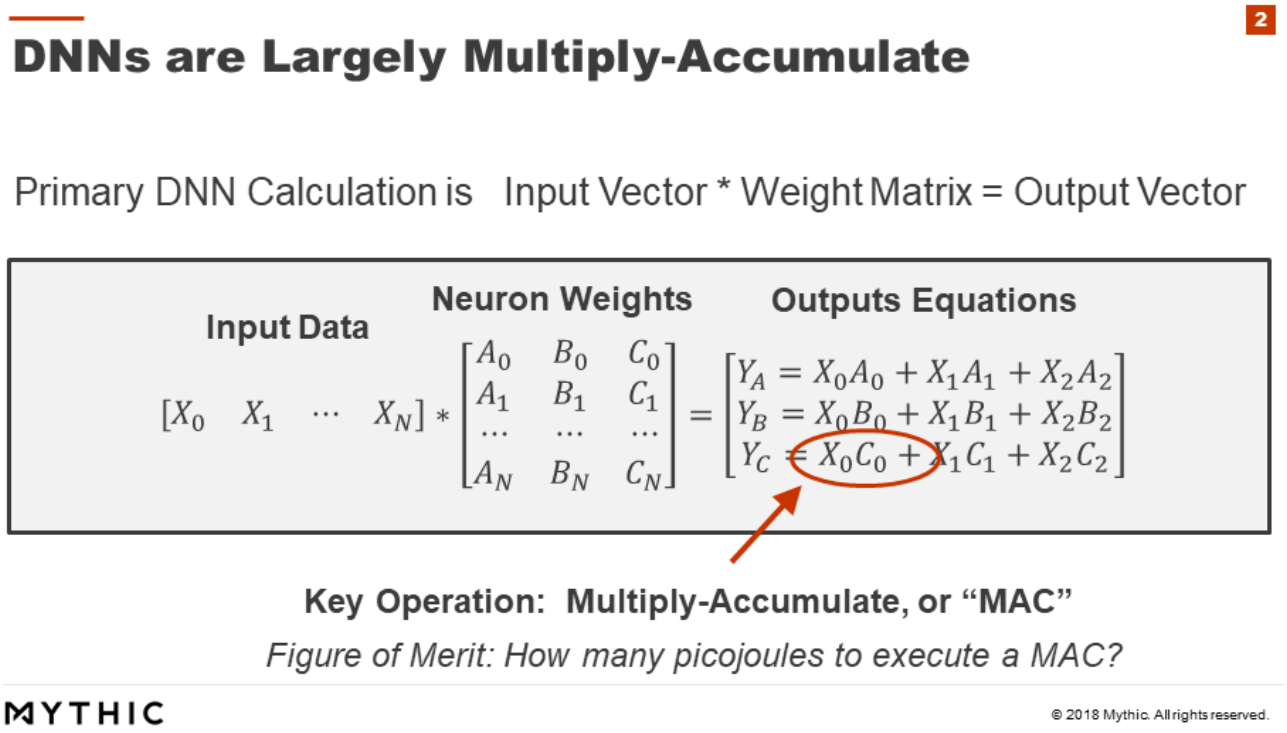
Mythic @ Hot Chips 2018

Dave Fick Follow
Aug 23, 2018 · 14 min read

We had a great presentation at Hot Chips 2018. In it, I discussed the motivation around our system and how it works. For those who were not able to attend, the slides are below with descriptions. There is also a video here.





To understand why we made our system choices, we first start off by discussing what neural network calculations are comprised of: matrix multiplies. The structure of a neural network is a graph of neuron groups flowing from one to the next. The outputs of one group of neurons are the inputs of the next group of neurons. To run the neural network calculations, we need to multiply the neuron inputs by the neuron weights; the inputs are a vector and the neurons are a matrix. The outputs are the vector created by multiplying these two together.

The matrix multiply is comprised of many smaller multiply-accumulate (MAC) operations, which are simple Y=Y+A*B operations. In fact, there are so many of these calculation performed in a neural networks, they completely dominate the other functions performed like ReLu and MaxPool.

The dominance of MACs in neural networks leads to a few things:

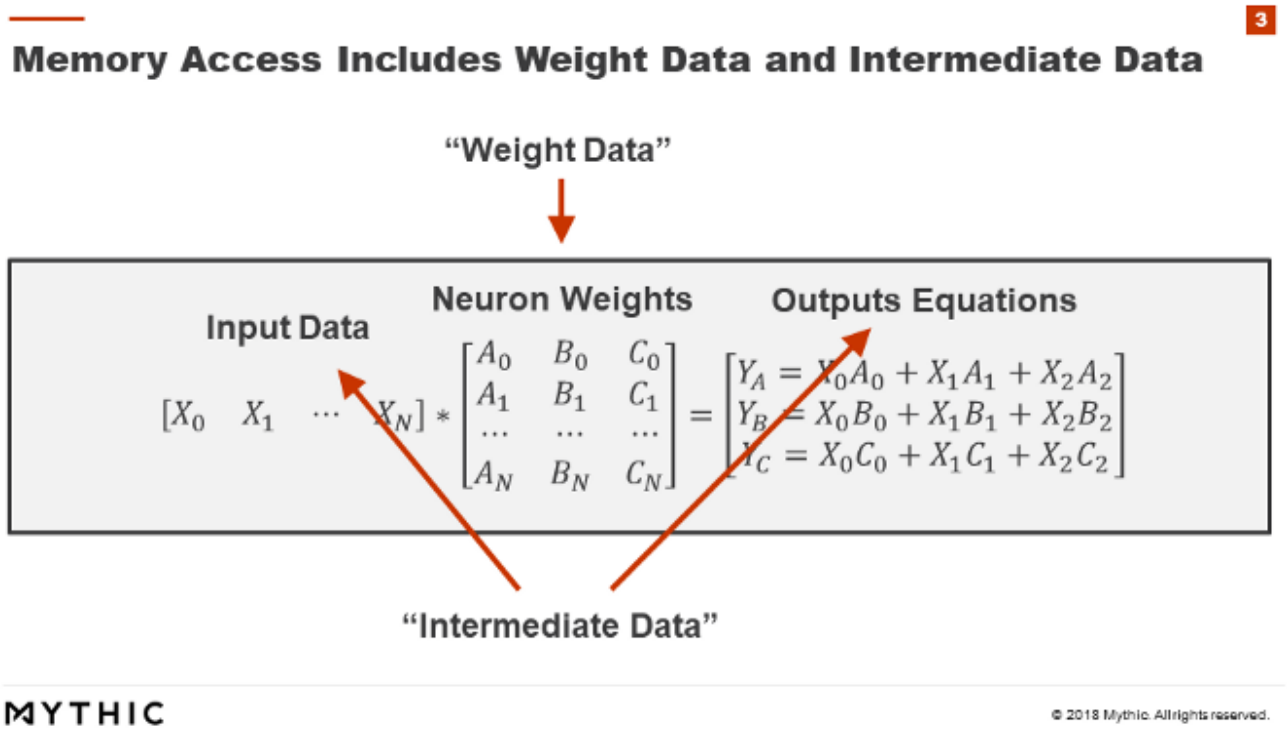
1. The complexity of a neural network can be expressed as the total number of MACs (e.g., 10 billion)
2. The performance of a system can be expressed in terms of TMACs/S (trillions of MACs per second)

3. The efficiency of a system can be expressed in terms of pJ/MAC (picojoules per multiply-accumulate operation)

Putting those together, if you know the pJ/MAC of a system, the number of MACs to calculate a neural network, and the frame rate that you want to run at, you can calculate the power that the system will draw.

Example:
(10 pJ/MAC) * (10 billion MACs/frame) * (30 frames/second) = 3 watts (joules/second)

In edge applications, there is a power and/or thermal limit (e.g., 1 watt). In many edge systems, these end up limiting performance before the actual processor performance does. As a result, the efficiency metric often directly translates to performance since additional efficiency affords additional headroom for more performance (2x as efficient => 2x more performance).



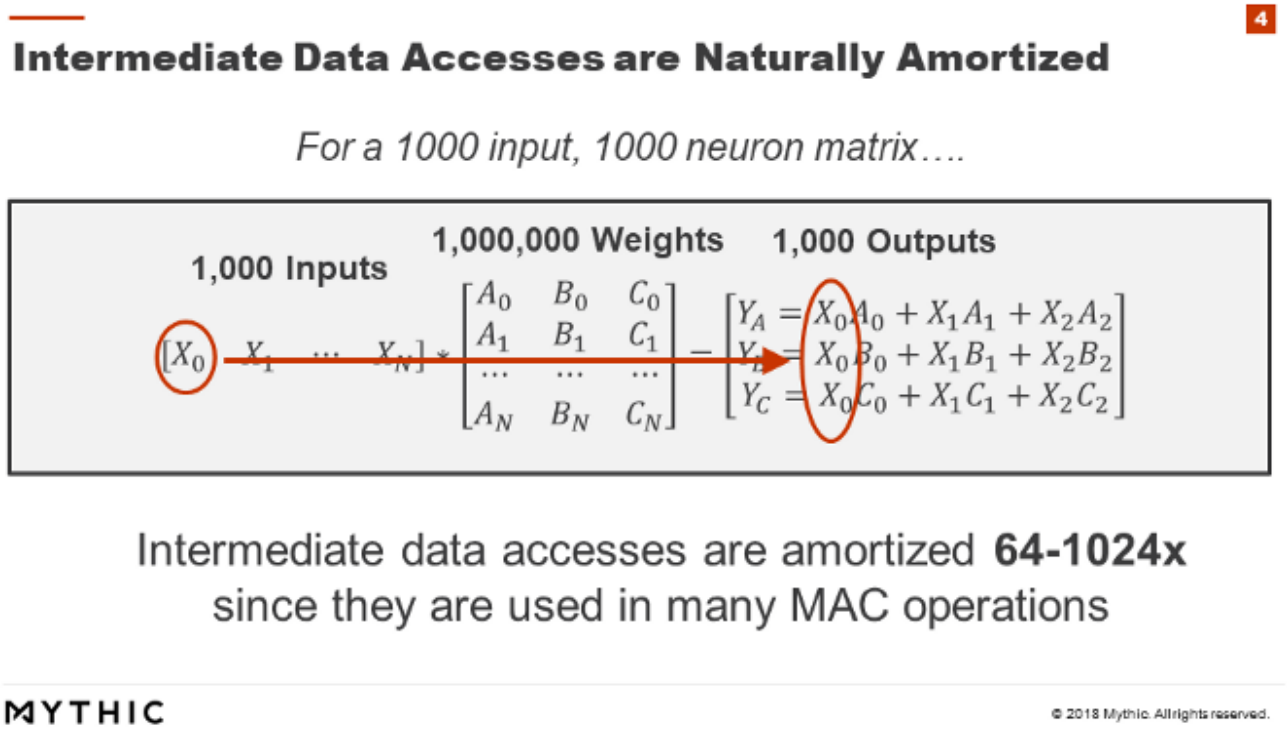
But… what I just said was accidentally misleading. Focusing on pJ/MAC makes it sounds like efficiently executing a multiply-accumulate operation is energy intensive, but it is not. In fact, the energy for that is very small.

In reality, systems get bogged down not by the MAC operation itself, but from memory accesses and interconnect needed to support those MAC operations. The neuron weights and the input data need to get to the MAC unit, from SRAM or DRAM, by reading that memory and then transmitting across the chip. The memory access and interconnect energy consumption are what comprises almost all of that pJ/MAC figure.

To analyze this further, we can partition the memory accesses into two groups:

- **Weight data:** these are the parameters that are developed during the training process. These are static during inference
- **Intermediate data:** this is the input to the neural network, the data passsed between the groups of neurons, and the output of the neural network. This changes for each neural network computation.

These two groups have different properties, which we will show.



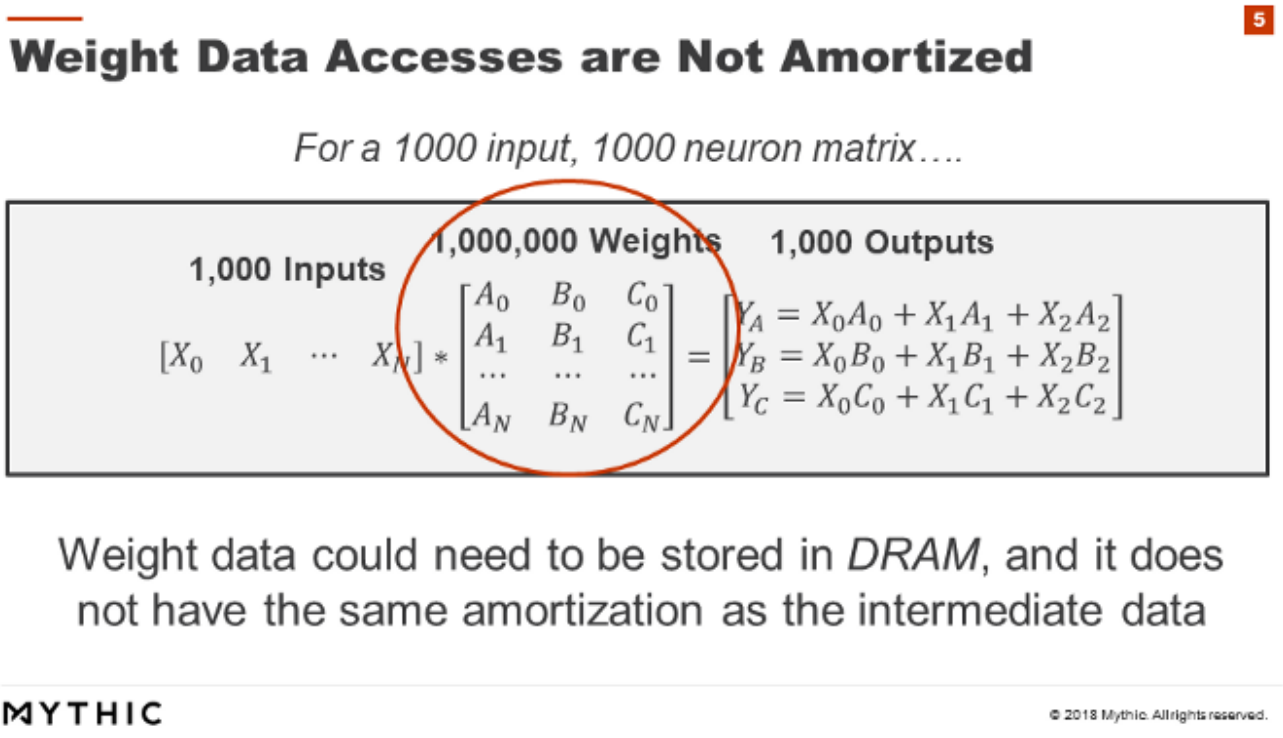
To illustrate these properties better, we consider a neural network stage with 1000 inputs and 1000 neurons. The input data is a vector of 1000 elements (for the 1000 inputs) and the output is a vector of 1000 elements (results of the 1000 neurons). The

weights, on the other hand, are a matrix of 1 million elements! This is 500x larger than the inputs and outputs *combined*.

In addition to the data size difference, there is also a major usage difference, which can be seen with the highlighted element X0. When we read X0 (likely from SRAM), we pay an energy cost (e.g., ~1 pJ for an 8-bit value in 40nm). That cost is amortized across all of the MACs in which it is used; for the input data in the example this is 1000 MACs. Therefore the read of X0 only contributes 0.001 pJ/MAC to our energy consumption.

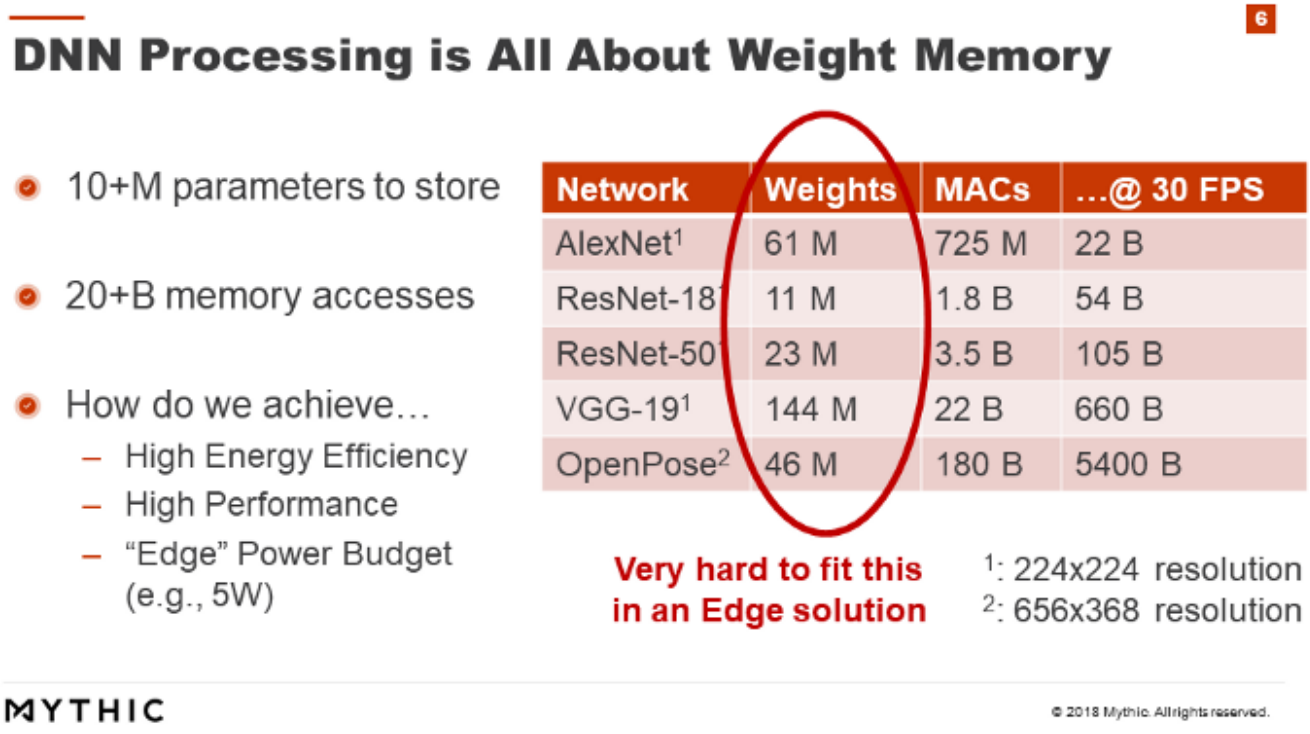
Typically, the number of neurons in a stage ranges from 64–1024, and the amortization of accessing intermediate data has a similar factor.

(Note, there are going to be multiple intermediate data accesses, I am not giving a full account here to keep the explanation concise.)



Each weight element is only used once, however. As a result, the energy cost of reading the weight is directly added to the cost of our MAC. Between the two, we are now at 1.001 pJ/MAC, with the vast majority of the contribution from reading the weight data from SRAM.

The weight data may even be coming from DRAM, which would then make the energy cost very high!



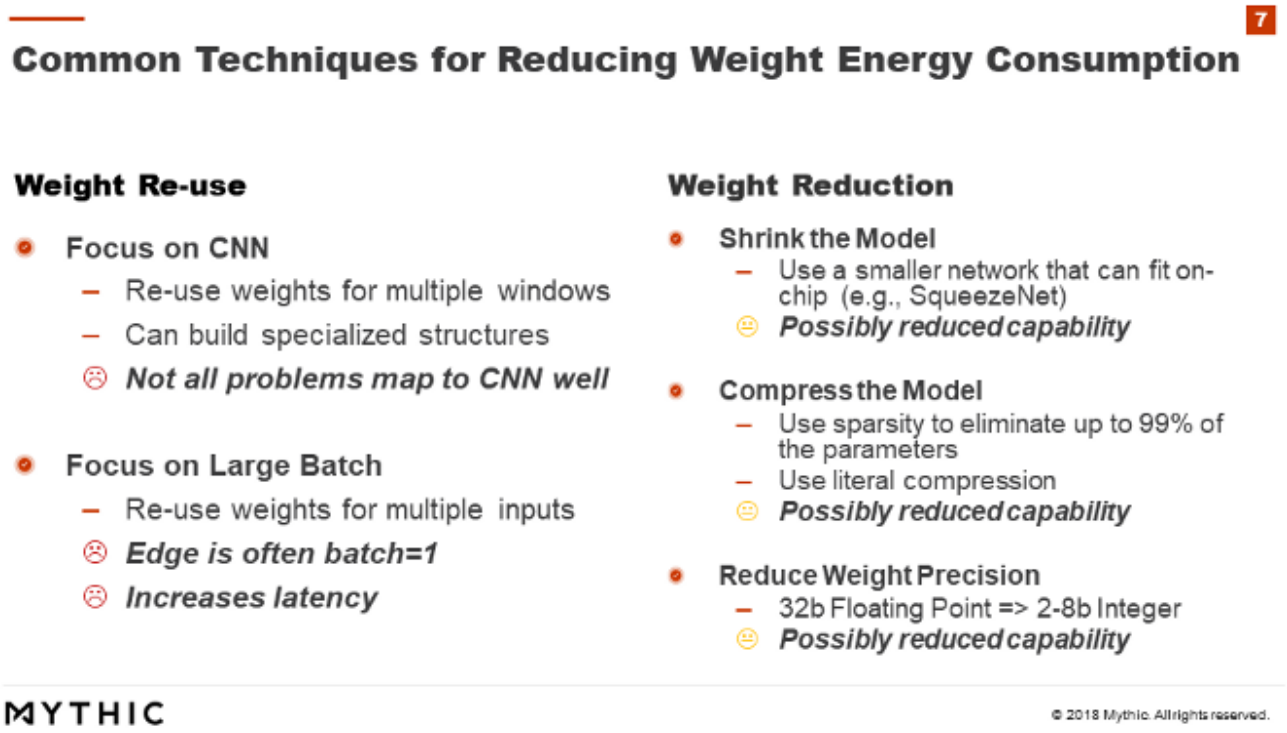
Because of these properties, designing an Intelligence Processing Unit (IPU) is all about managing these weights. As you will see later, major architectural decisions revolve around these decisions.

On this slide, some major neural networks are shown, including the number of weights and the number of MACs (and the number of MACs at 30 FPS). When looking at the number of MACs to execute a network, do not think of a MAC as a multiply and accumulate, think of it as weight memory access.

For many of these networks, even for tiny 224x224 input resolution, the number of MACs that we need to perform are in the billions. Going to 1080p or 4k will push these numbers into the trillions.

This is complicated by the number of parameters, which cannot even fit in SRAM in a typical Edge system. Therefore, we *must* keep some parameters in DRAM.

So how do we do this with an Edge power budget of 5W, while achieving the performance that we need?



We need to reduce the energy consumption of accessing these weights. There are two categories for accomplishing this: **weight re-use** and **weight reduction**.

Weight re-use means that we will try to use a weight multiple times each time it is accessed, similar to how the intermediate data is re-used.

The first way to do this is to focus on convolutional neural networks. These use a set of neurons in a windowing operation and use multiple subsets of the input data to the neurons. The larger the image, the larger the number of positions that will be fed in and the the greater of the amortization.

Using large batches provide a similar benefit. We can process multiple input frames in parallel using the same network, and can amortize the access to those weights.

Weight reduction means that we will try to reduce the number of weights in the network.

The first way to do this is to reduce the size of the model we are using, as much as possible. Networks like SqueezeNet push this to the extreme.

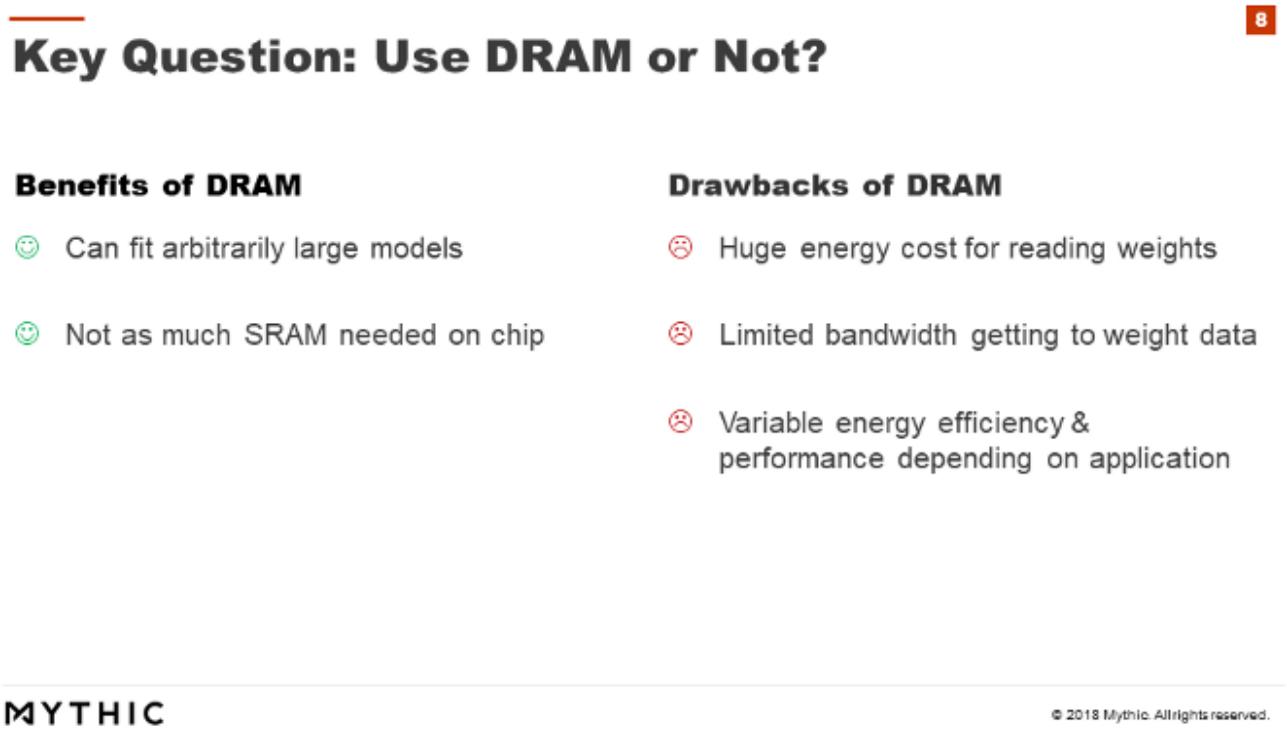
The next way to do this is to compress the weights. Taking advantage of sparsity is a great way to do this, and the sparsity can be intentionally increased during the training

process. Sparsity makes many of the weights equal to zero, as high as 99%.

Finally, the precision of neural network weights can be reduced from 32b down to 8b (the gold standard for inference) or even lower.

Each of these techniques has a cost, however. On the weight re-use side, not every application supports these strategies. In particular, I believe recurrent neural networks will become more popular over time as they add the concept of time which is important to video and audio applications.

On the weight reduction side, using these techniques will reduce the accuracy of the network to varying degrees. They can help, but they are by no means free.

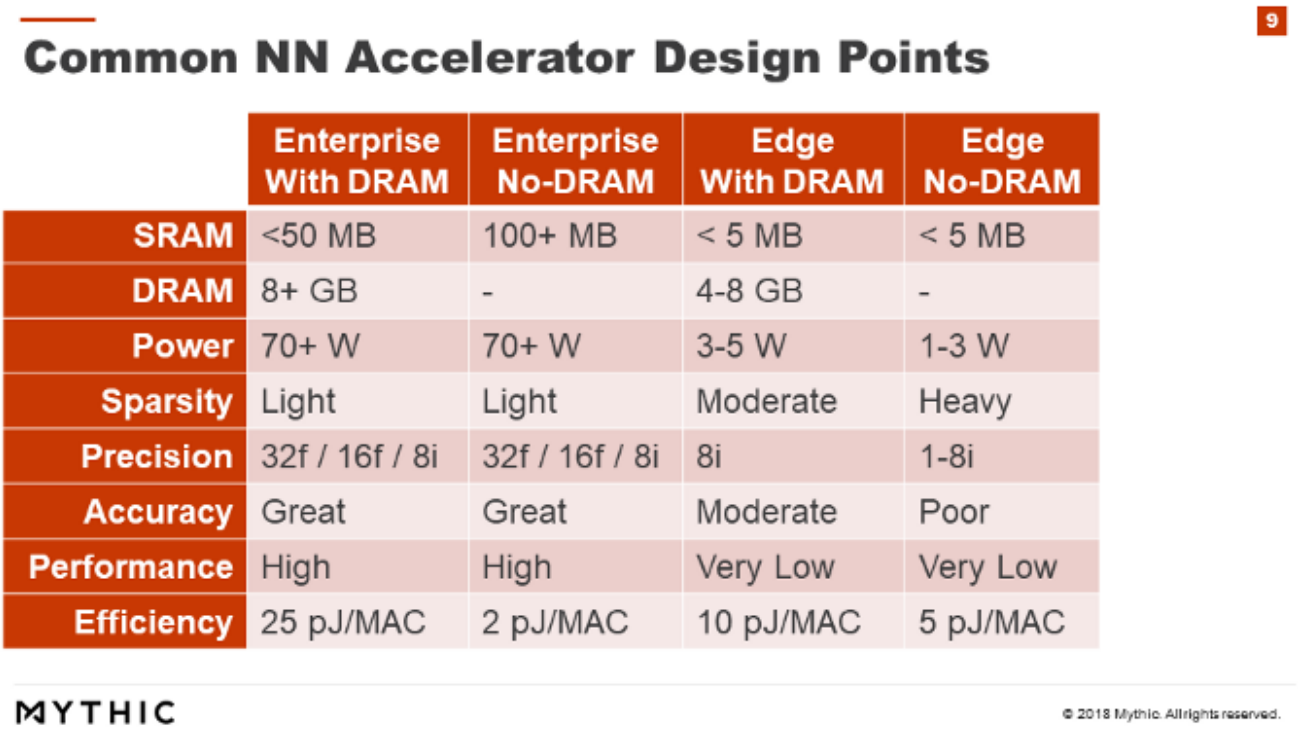


When designing a new system, we need to start with a key question: Should we use DRAM or not? There are some pro's and con's.

Perhaps the easiest way to think about this choice is to consider the implications of NOT using DRAM. Without it, the size of the model we can run is limited by the amount of SRAM that we have available on the system, so we better have a lot of it, and even then we have a restriction. However, the cost of reading weights is nearly constant (always

SRAM) and low (compared to DRAM), and the bandwidth getting to the SRAM is huge compared to what is available through DRAM.

If we choose to go with DRAM, then we have no restrictions on model size, and we do not need a huge amount of SRAM. But, we now have higher energy to access DRAM, which varies since each application will have a different degree of dependence on DRAM.



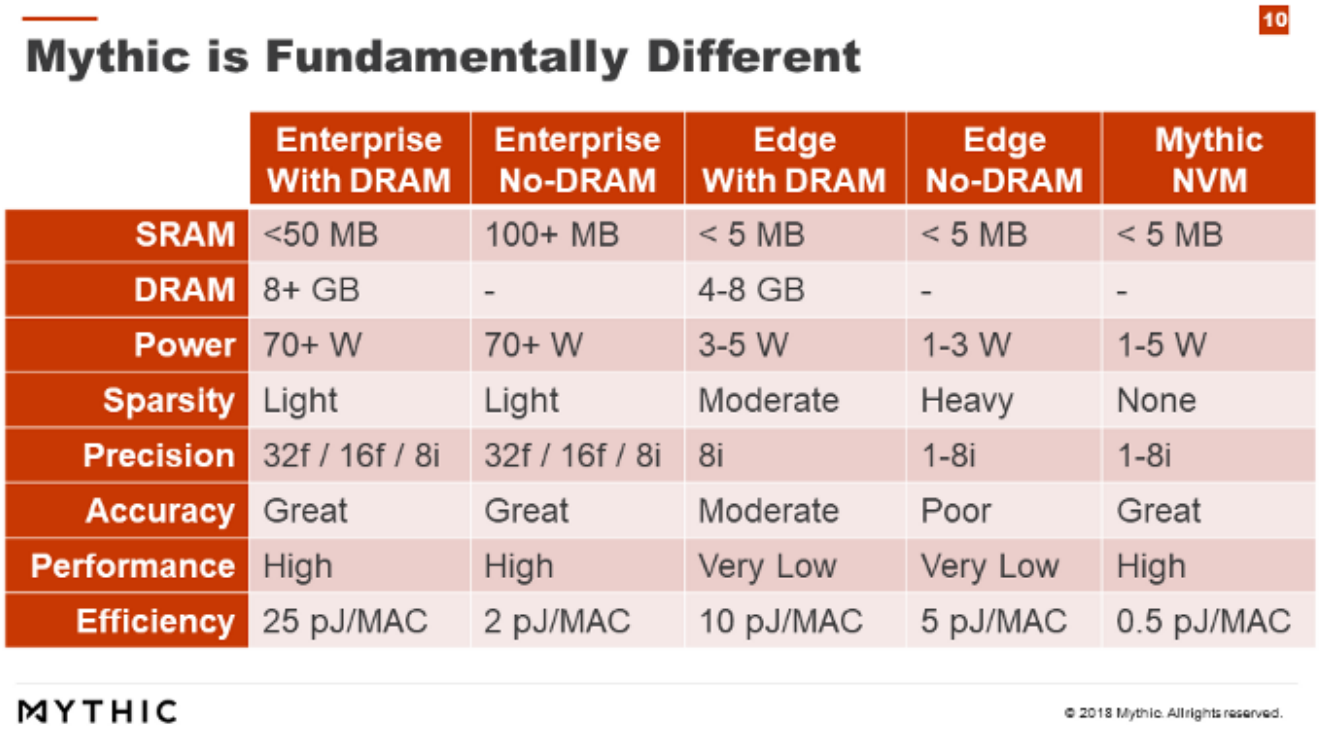
	Enterprise With DRAM	Enterprise No-DRAM	Edge With DRAM	Edge No-DRAM
SRAM	<50 MB	100+ MB	< 5 MB	< 5 MB
DRAM	8+ GB	-	4-8 GB	-
Power	70+ W	70+ W	3-5 W	1-3 W
Sparsity	Light	Light	Moderate	Heavy
Precision	32f / 16f / 8i	32f / 16f / 8i	8i	1-8i
Accuracy	Great	Great	Moderate	Poor
Performance	High	High	Very Low	Very Low
Efficiency	25 pJ/MAC	2 pJ/MAC	10 pJ/MAC	5 pJ/MAC

Most systems cleanly fit into four categories broken down by Edge/Enterprise and DRAM/No-DRAM.

On the Enterprise front, we can heavily leverage batch size and include a huge amount of SRAM on the system. In particular, the No-DRAM case is supported by including 100's of megabytes of SRAM, which allows it to achieve the best typical energy efficiency

Enterprise systems are huge, though, often taking a full reticle and drawing 70–200+ watts. Clearly this would not work in most edge applications, like a smart camera system. A device like the Nest IQ relies on the casing to dissipate heat, and it has limited surface area — if you touch one, you will find that it is already very hot.

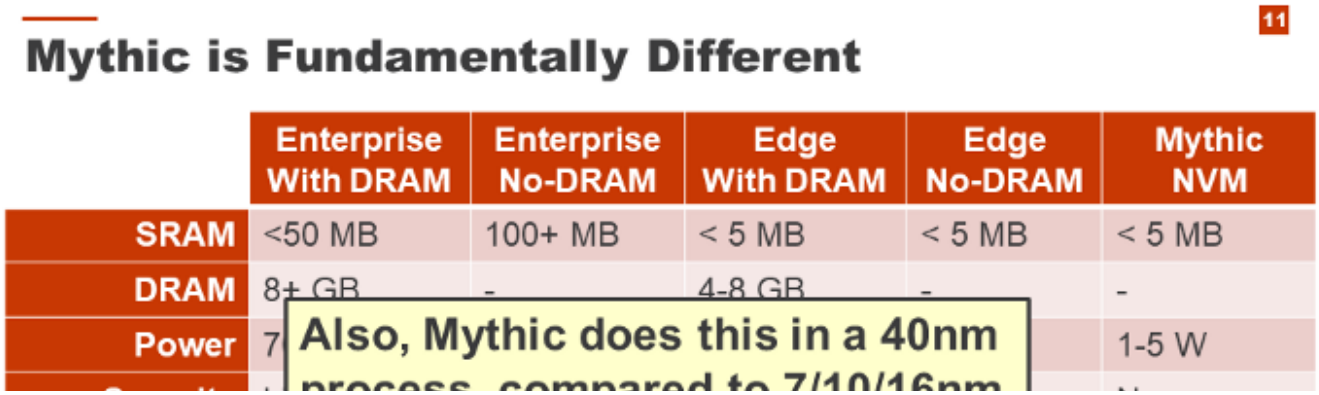
On the Edge front, a huge amount of SRAM is out of the question. For these systems to achieve No-DRAM, they need to heavily leverage the Weight Reduction techniques, which takes its toll on the achievable accuracy. Even with the inclusion of DRAM, these techniques are often dialed-up to fit within the edge power budget and to hit performance targets.



	Enterprise With DRAM	Enterprise No-DRAM	Edge With DRAM	Edge No-DRAM	Mythic NVM
SRAM	<50 MB	100+ MB	< 5 MB	< 5 MB	< 5 MB
DRAM	8+ GB	-	4-8 GB	-	-
Power	70+ W	70+ W	3-5 W	1-3 W	1-5 W
Sparsity	Light	Light	Moderate	Heavy	None
Precision	32f / 16f / 8i	32f / 16f / 8i	8i	1-8i	1-8i
Accuracy	Great	Great	Moderate	Poor	Great
Performance	High	High	Very Low	Very Low	High
Efficiency	25 pJ/MAC	2 pJ/MAC	10 pJ/MAC	5 pJ/MAC	0.5 pJ/MAC

Mythic introduces a new option which uses non-volatile memory to take on the challenges of weight data. This system does not need DRAM, achieves excellent accuracy and performance, and achieves best available energy efficiency.

Mythic is essentially achieving enterprise-level performance characteristics within an edge form factor.

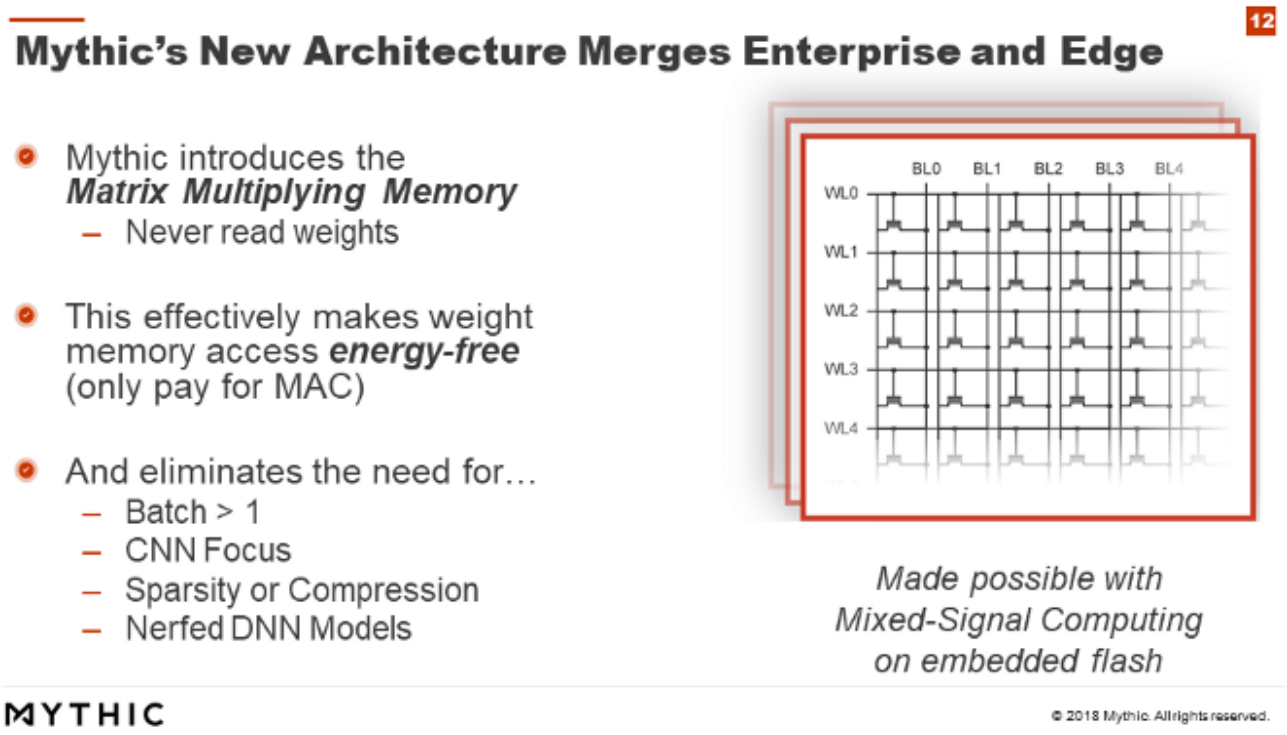


	Enterprise With DRAM	Enterprise No-DRAM	Edge With DRAM	Edge No-DRAM	Mythic NVM
SRAM	<50 MB	100+ MB	< 5 MB	< 5 MB	< 5 MB
DRAM	8+ GB	-	4-8 GB	-	-
Power	7				1-5 W

Sparsity	L[process, compared to 7/10/16nm				None
Precision	32f / 16f / 8i	32f / 16f / 8i	8i	1-8i	1-8i
Accuracy	Great	Great	Moderate	Poor	Great
Performance	High	High	Very Low	Very Low	High
Efficiency	25 pJ/MAC	2 pJ/MAC	10 pJ/MAC	5 pJ/MAC	0.5 pJ/MAC

MYTHIC

Additionally, Mythic does this in a 40nm process, while these other systems are in much newer process nodes. *Effectively, Mythic has turned back the clock on process scaling.* While other system designers are struggling to get from 7nm to 5nm, Mythic will be scaling to 28nm.

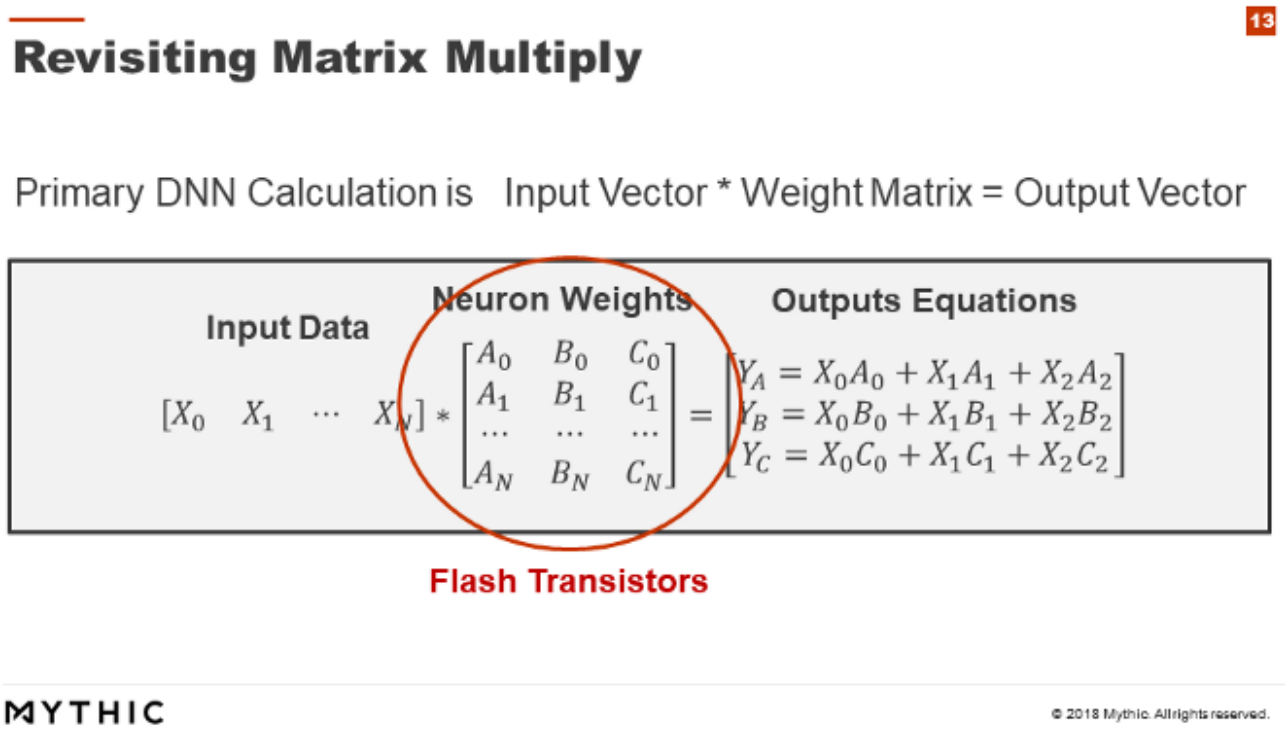


Mythic accomplishes these feats by introducing a Matrix Multiplying Memory. This memory structure is rarely read like a standard memory. Instead, we provide it a vector of input data and it generates a vector of output data. From an energy cost perspective, we never pay the energy of reading the weights. Instead, we only pay the energy cost of performing the MAC operations.

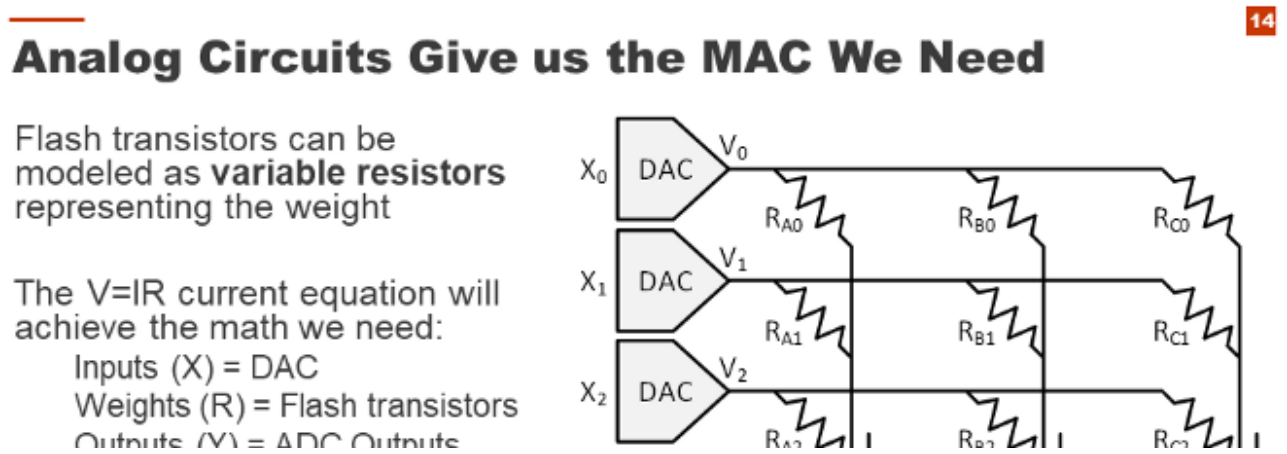
This memory accomplishes this by using a multi-level cell non-volatile memory (MLC NVM) with analog computation. For our first product we are using embedded flash since

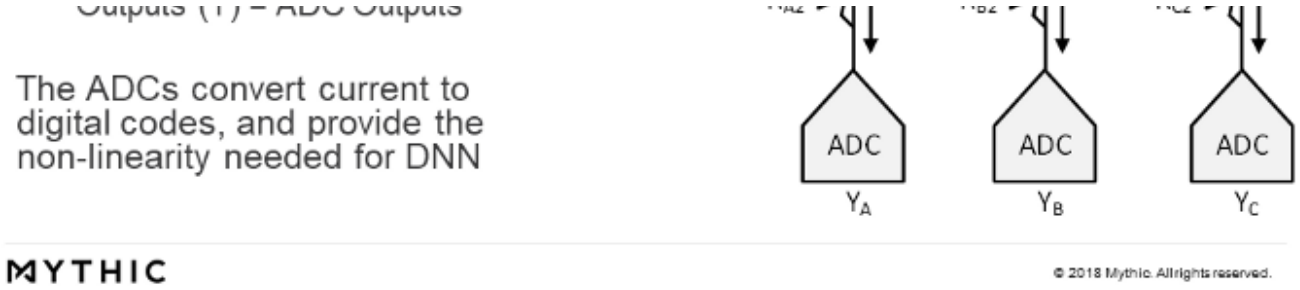
it is commercially available today. In the future, there are many opportunities including RRAM, Phase Change Memory, Carbon Nanotube Based Memory, and more.

As a result, we do not need to use many of those Weight Re-use and Weight Reduction techniques that we discussed earlier. In particular, we can use Batch=1 which is good for Edge, we can support RNN as easily as CNN, we do not need to use Sparsity or Compression, and we do not need to nerf our DNN models.



Re-visiting our equations from earlier, we are storing the neuron weight matrix in flash transistors. Each of the weights is represented as an 8-bit (256 level) value in the flash array.



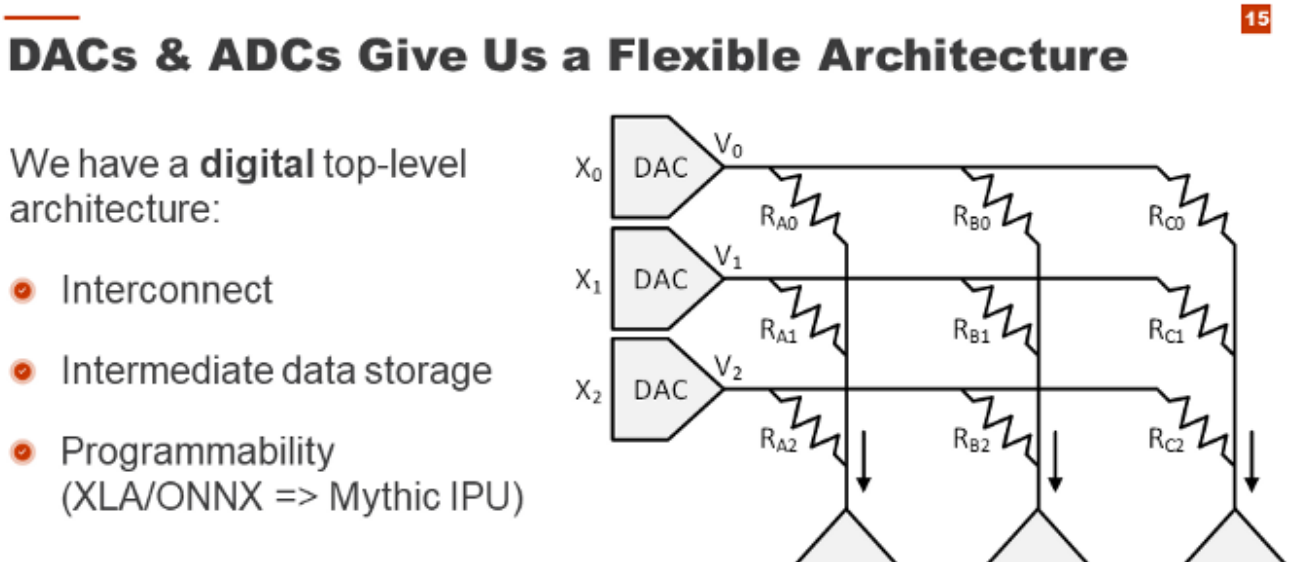


The flash transistor is acting like a variable resistor. We can program the flash cell to achieve a particular resistance value (which can be refreshed from time-to-time by off-chip true values).

The resistor performs the MAC operation through the $V=IR$ current equation, which can be re-written as $I=VG$, where $G = conductance = 1/R$. The inputs are therefore voltages, which we generate using digital-to-analog converters (DACs). The outputs are currents which we convert back to digital values using analog-to-digital converters (ADCs). To complete the MAC operation, we need to *accumulate*, which we accomplish by connecting columns of resistors together. The ReLu or other non-linearity is provided by the ADCs.

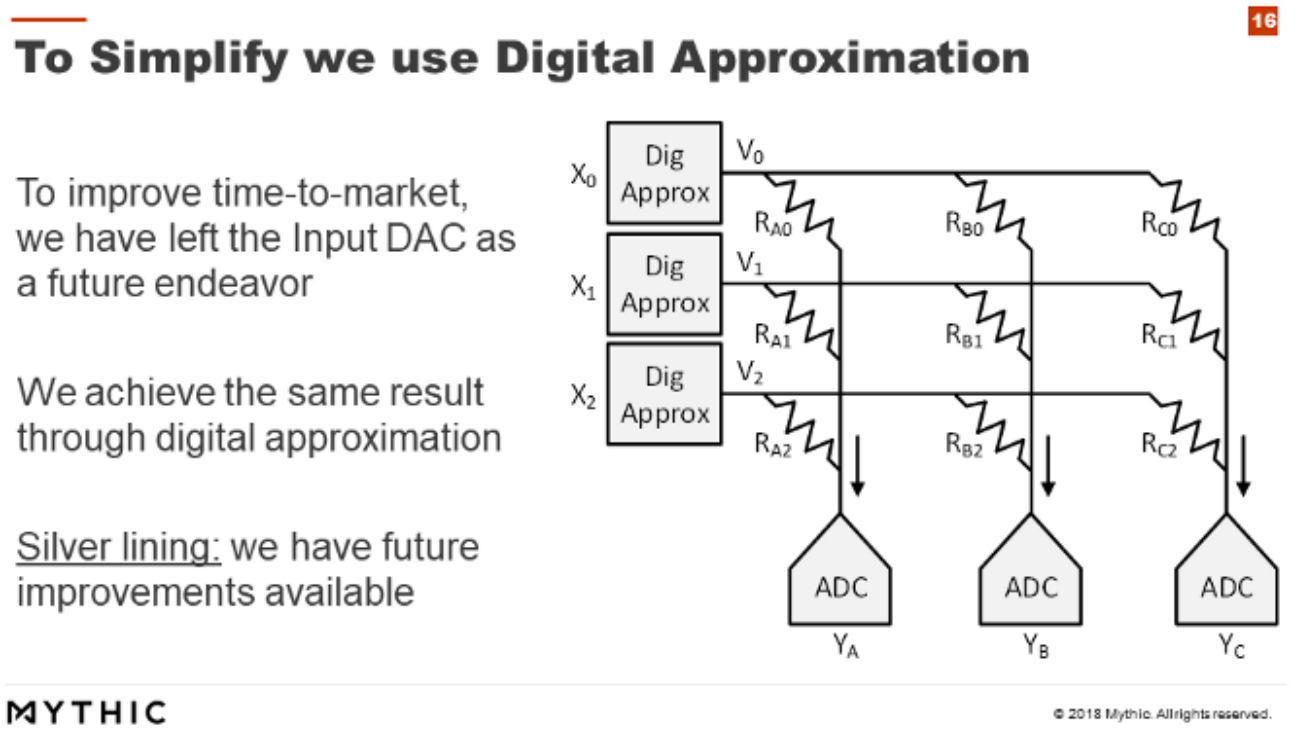
To use this memory structure, we never read the values out of the array. Instead, we apply our input vector through the DACs, and we read the output vector from the ADCs. Everything is done in parallel, which also gives us a huge amount of performance in addition to the great energy efficiency.

As noted earlier, this technique works with many MLC NVM types.



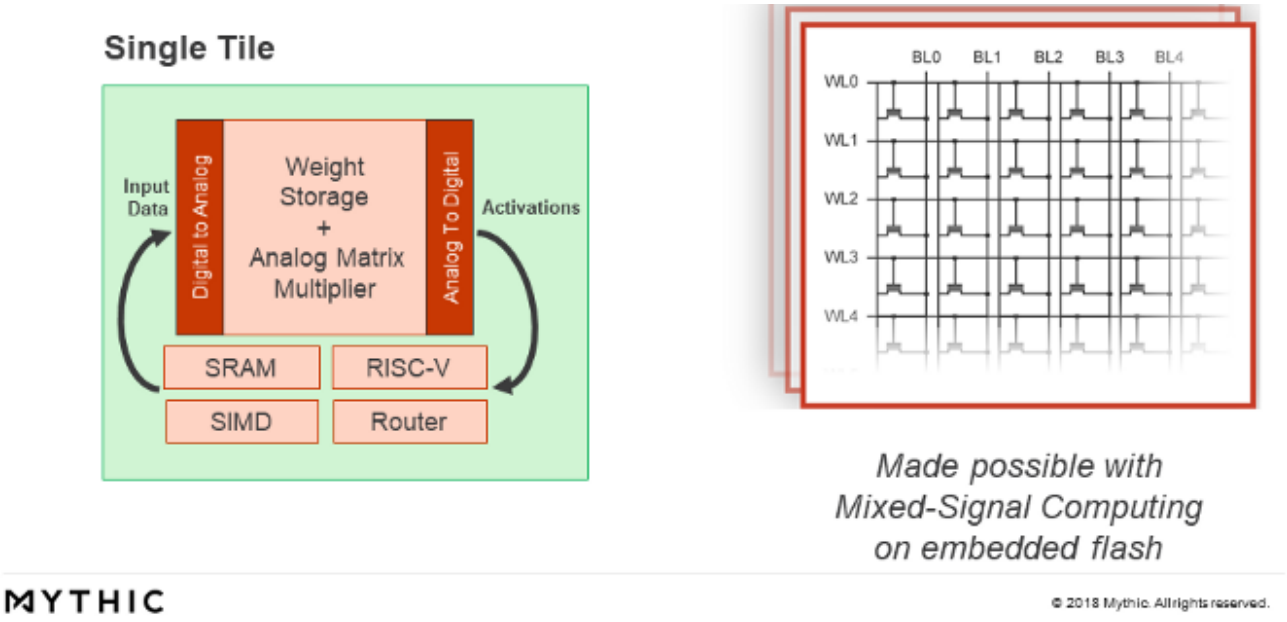
Although we leverage analog computing to perform the MAC operations, we do not think analog is great at everything. Digital is very good at most computer architecture problems, including interconnect, data storage (via SRAM), and reconfigurability.

For reconfigurability, we plan to support formats like XLA and ONNX, which are standards for expressing trained neural networks.

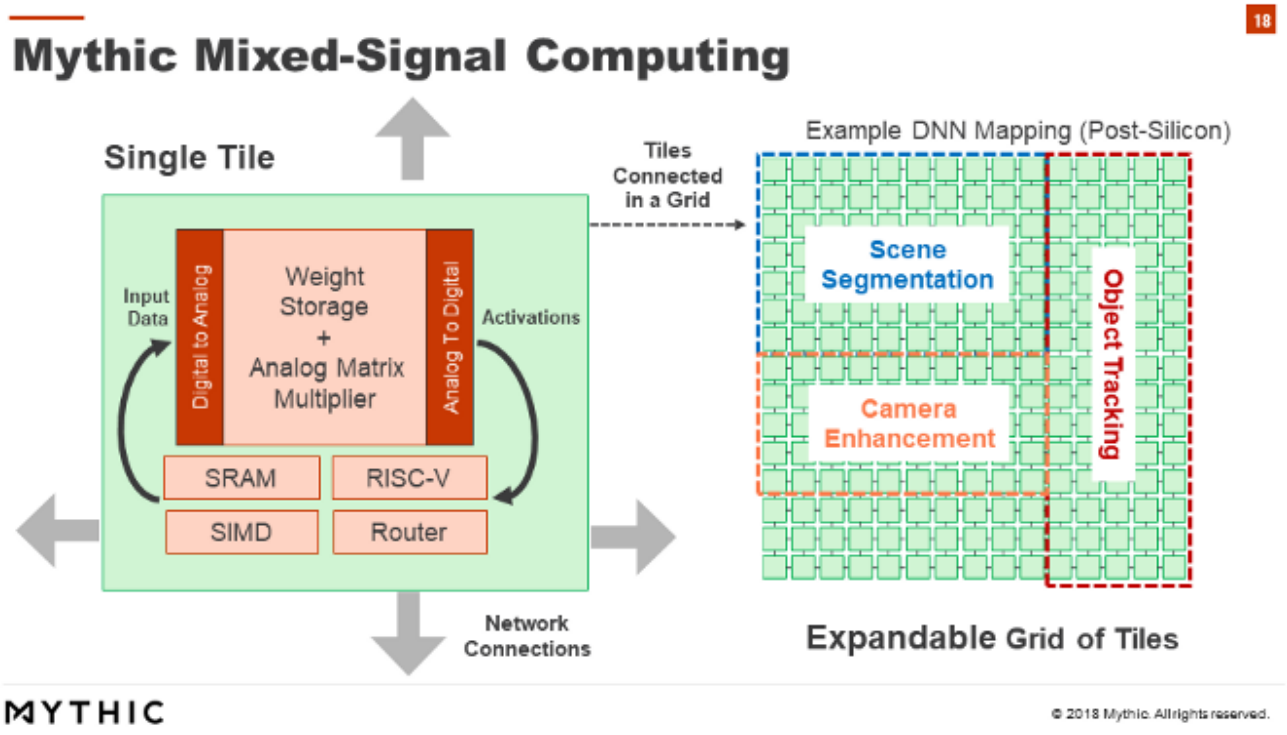


To improve our time-to-market, we are using a digital approximation technique to skip creating the DACs for our first product. Similar to how you multiply numbers by hand on paper, we are computing each input bit separately and then adding the results together.

This means that we have a big improvement available without even using process scaling.



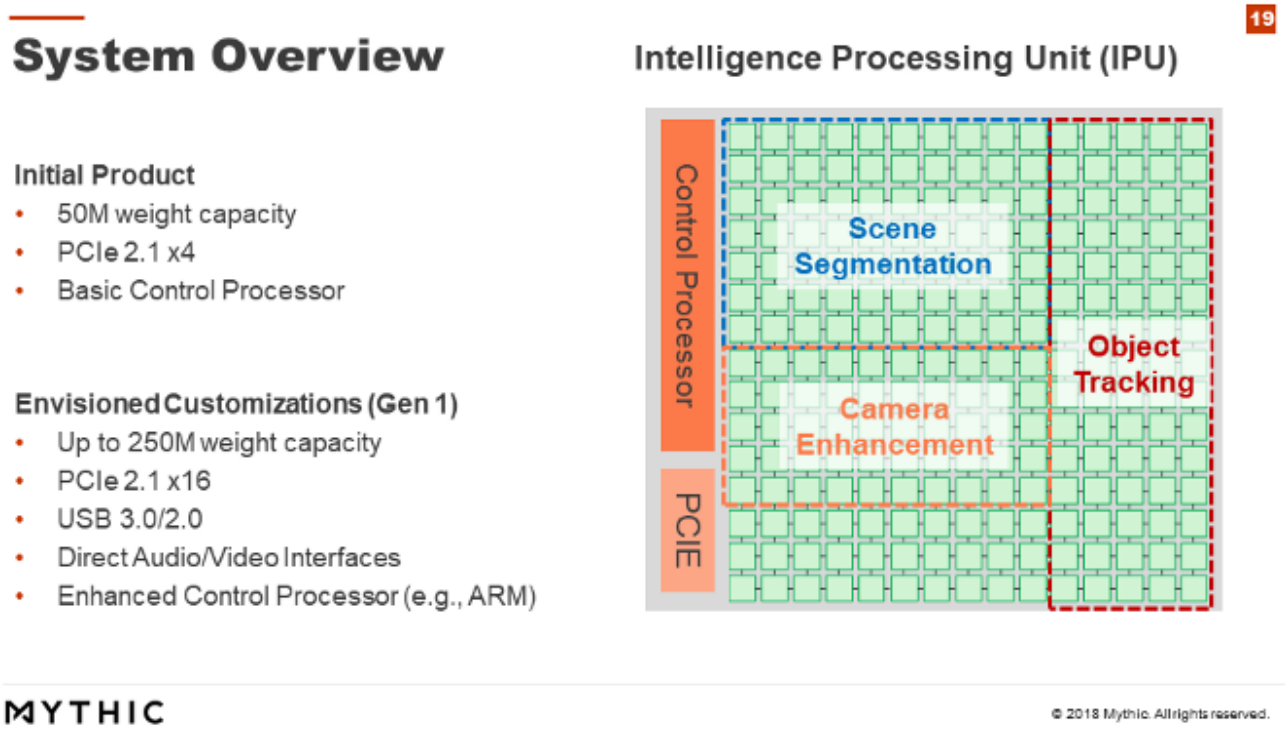
Our system is a tile-based design, where the large green box represents the tile. Inside of it we have a Matrix Multiplying Memory and the digital control logic mentioned before: SRAM for storing intermediate data and program code, a SIMD engine for performing MaxPool and other non-MAC operations, a RISC-V processor to provide flexible control logic, and a network-on-chip (NoC) router to provide the global interconnect.



The tiles are laid out in a grid, and the NoC routers provide connections between adjacent tiles. The size of the grid can vary between products, which allows us to provide a variety of performance and cost points for our customers.

The use of flash memory means that reprogramming the chip takes some time; we are targeting about one minute for our first product. The introduction of new, rapidly rewritable NVM technologies could change this in the future. We can also support multiple applications by partitioning the system into separate regions.

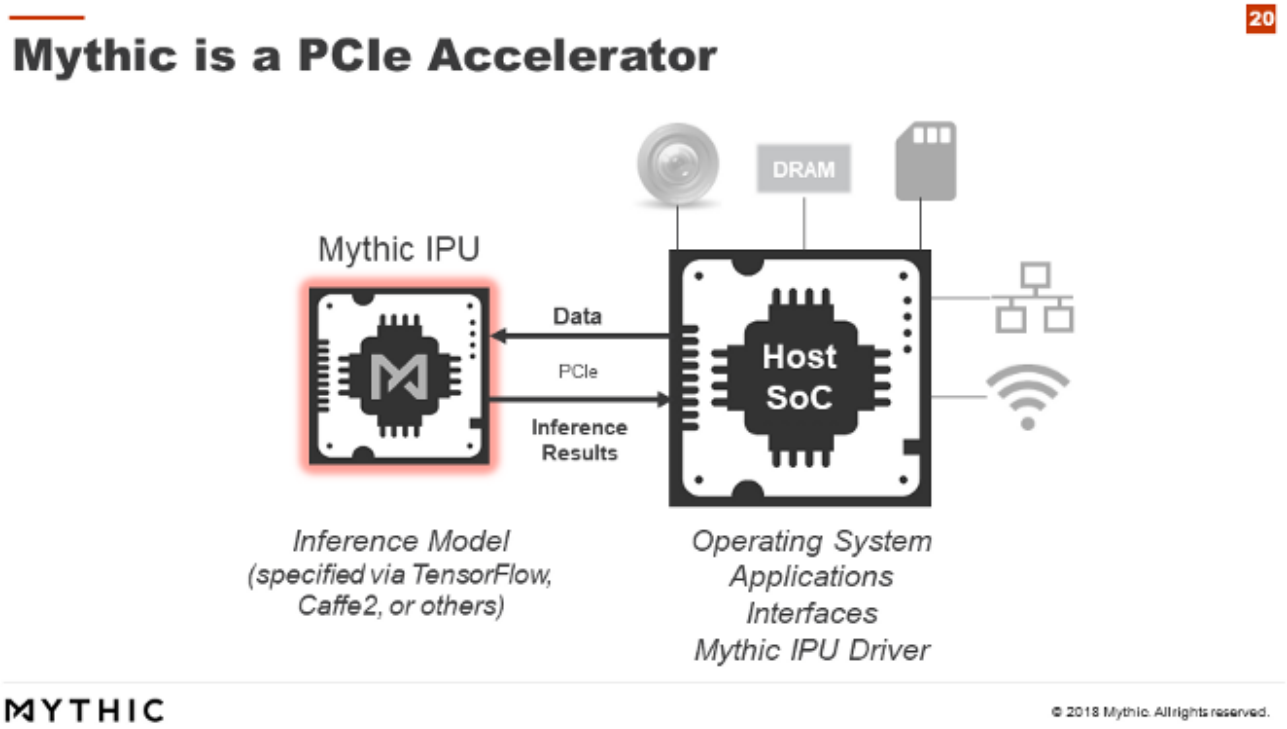
Typically, in most edge systems there is not a need to switch between applications regularly. For instance, an autonomous vehicle will run the same vision processing stack until it receives an update, which may be once a day or once a week.



For our initial product, we are targeting a 50M weight capacity, PCIe 2.1 x4, and a basic control processor. The weight capacity translates to neural network storage space as well as performance, similar to an FPGA. The PCIe 2.1 x4 port provides 20 gbps of bandwidth, which is sufficient to carry 4k video. The basic control processor interacts with the host and directs the tiles in the system.

From here, we can make additional variants of the system. We can increase the number of tiles up to a 250M weight capacity, increase the number of PCIe lanes, or even add

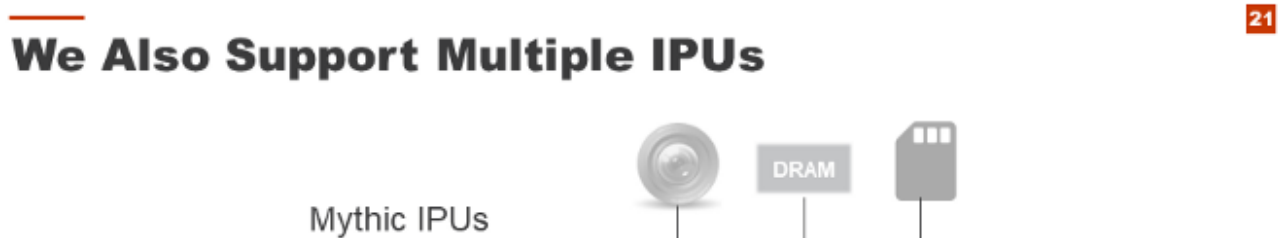
new interfaces. Additionally, we could add an ARM control processor to provide the ability to run local programs on the system.

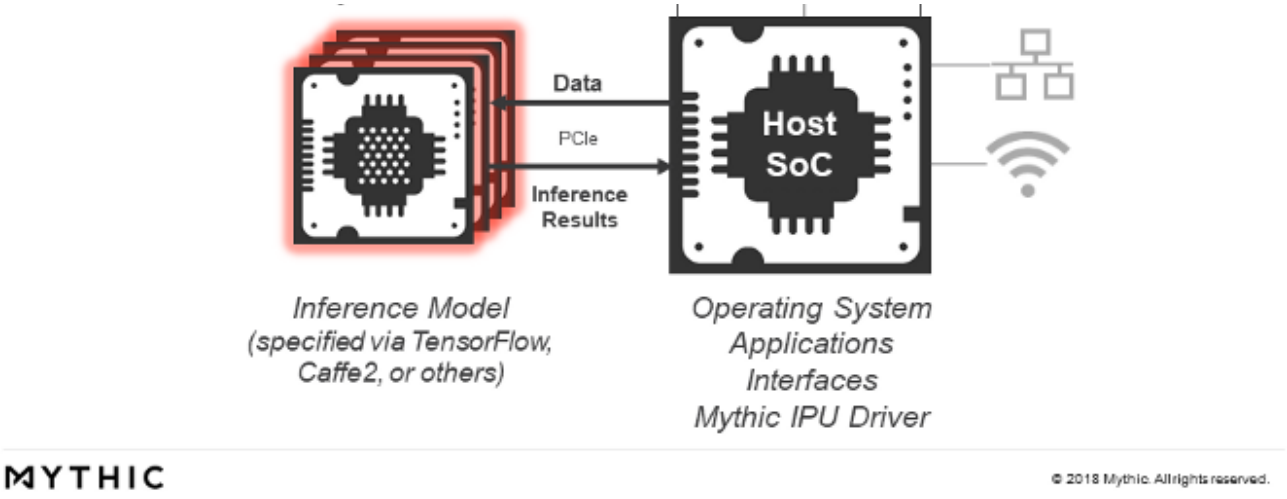


Our intelligence processing unit (IPU) integrates into the system by connecting to a host that has a PCIe port. In a smart camera system, the host is an SoC that connects to a camera, WiFi, and NAND flash.

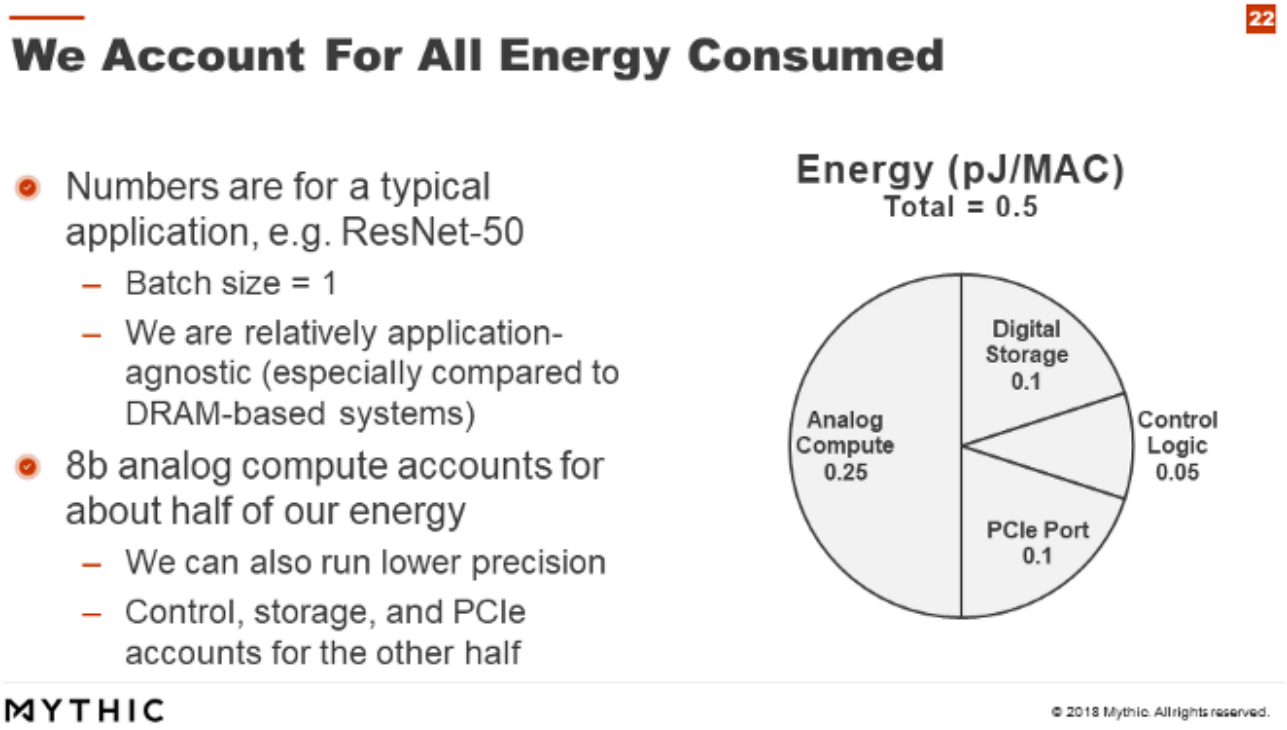
The host will set up the Mythic IPU by providing a firmware with an image of the inference model that is to be run. This firmware comes from our compiler, which received the network from TensorFlow, Caffe2, PyTorch, or other development environments.

When the host wants to run inference, it provides the input data over the PCIe port to the Mythic IPU, which will then return the inference results. During this time, there is no load on the host, other than providing the input data.





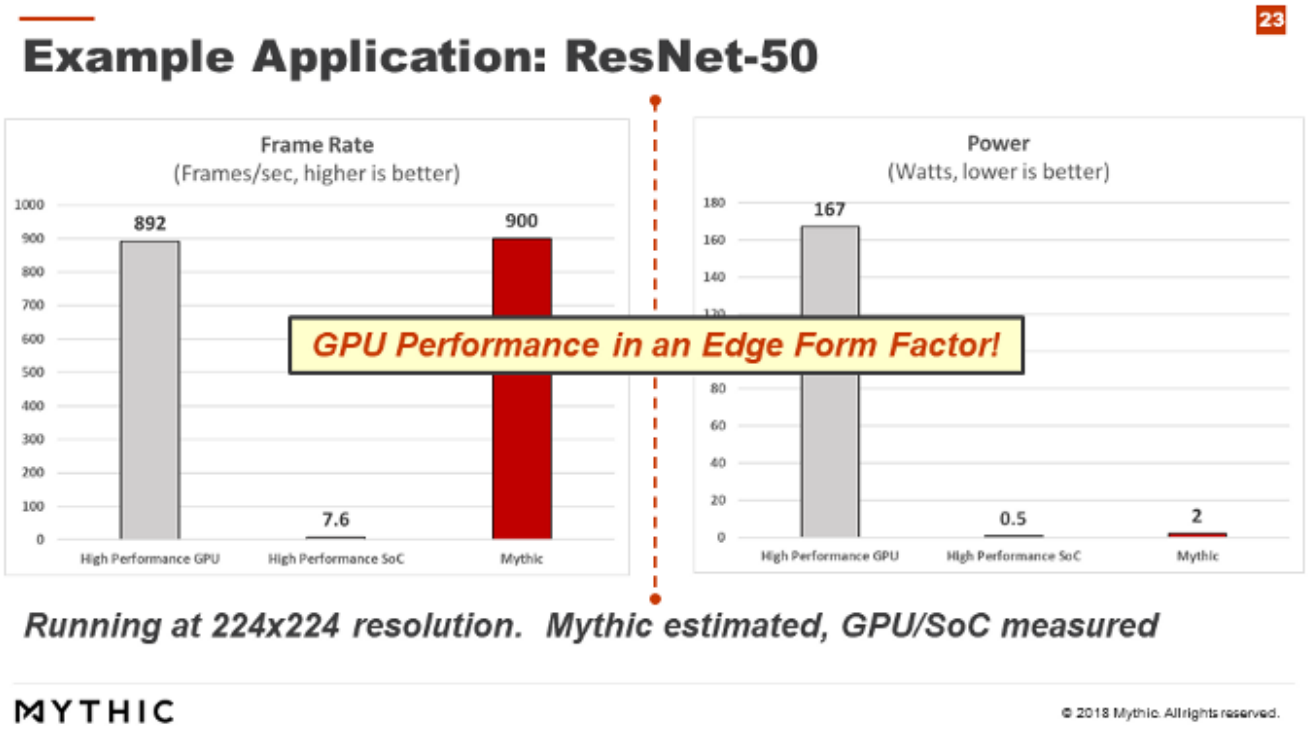
Some developers may want to run an application either with higher performance or more weights than what a single IPU can provide, so we support the ability to run multiple IPUs together, interconnected through the PCIe interface.

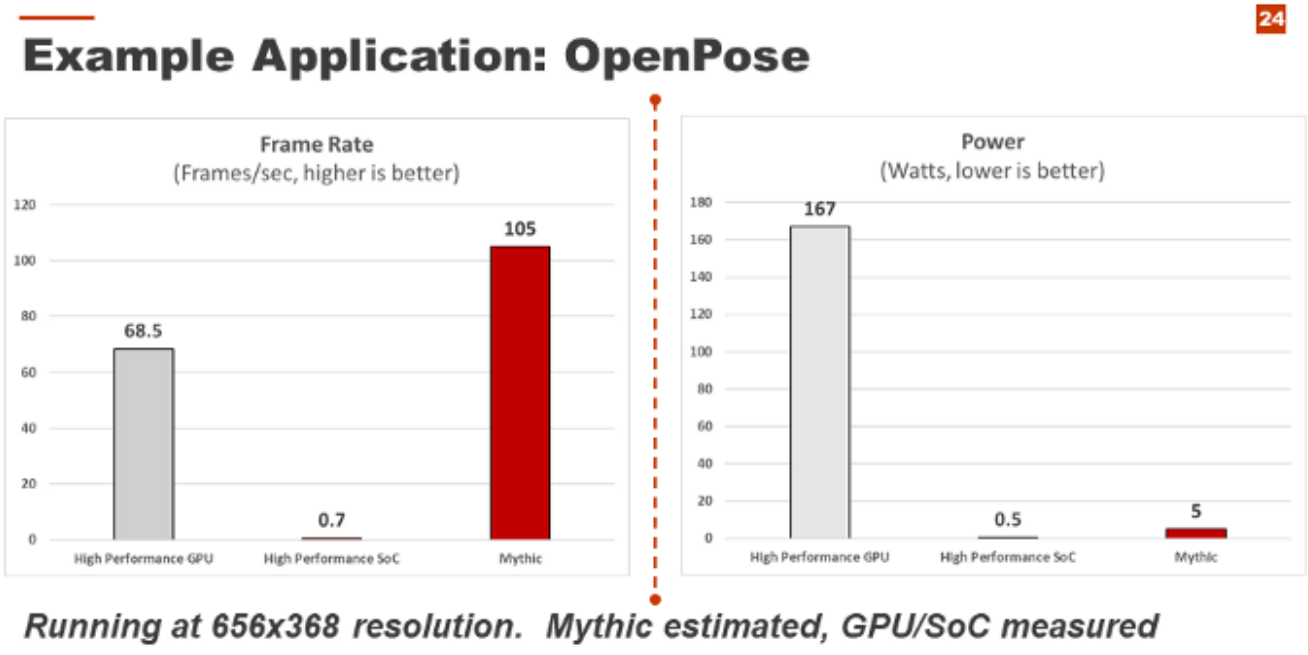


The energy figure that we provide is a full accounting of power in the system, including not only the MAC calculations themselves, but all of the managing digital architecture as well as the PCIe port. Nothing is left out (e.g., DRAM power or loading on the application processor).

Additionally, this figure is for a typical application, like ResNet-50 running with batch size of 1. It should not vary much one application to the next since the energy sources are largely agnostic to the data patterns.

This figure assumes that we are running in 8-bit mode (full accuracy for this system), but we can also run some stages at lower precision to save additional energy.





Above are some results for typical use-cases of our system.

Often, when we talk to an AI developer, they initially create their application on a high-performance GPU (e.g., 250W, $1000). They get the accuracy to the place they need, then they try to deploy to a high-end SoC. The result is that the performance craters to the point where they end up heavily stripping the application down, or changing the structure entirely (e.g., from RNN to CNN).

> What we provide is the ability to develop on GPU, then deploy at full performance on IPU, while staying within a reasonable power budget.

We are aiming to release alpha versions our tools at the end of this year to our early access customers. These are companies that are already neural network experts who are deploying real applications, but need 20–100x additional performance to get to where they want to be for their next product.

We will be releasing samples mid next year in the form of PCIe board, and providing volume shipments at the end of the year for both PCIe and BGA packages.

Please ask questions via Notes and I will create an FAQ below.

Artificial Intelligence Ipu Mythic Edge Computing Deep Neural Networks