



Programming Models for a Mixed-Signal AI Inference Accelerator

IWOMP 2020

Eric Stotzer, September 23, 2020.

© 2020 MYTHIC. ALL RIGHTS RESERVED.



Abstract

This talk will cover Mythic's hybrid mixed-signal computing architecture and unique software development tools, including a Deep Neural Network (DNN) graph compiler. In addition, some ideas will be proposed on how OpenMP can be used to program this type of architecture.

Mythic's Intelligence Processing Units (IPUs) combine analog compute-in-memory acceleration with digital processing elements. They are designed for high-performance and power-efficient AI inference acceleration.

The Mythic IPU is a tile-based dataflow architecture. Each tile has an analog compute array, flash memory for weight storage, local SRAM memory, a single-instruction multiple-data (SIMD) unit, and a control processor. The tiles are interconnected with an efficient on-chip router network.

Mythic has built a unique suite of development tools, including a DNN graph compiler, to enable the rapid deployment of AI inference applications on the IPU. The tools perform such actions as mapping DNNs to tiles, setting up dataflow conditions, and analog-aware program transformations.

Mythic, Inc.

- AI startup founded in 2012 focused on power-efficient AI inference processing
- Unique analog compute-in-memory (CIM) architecture using flash memory
- 100+ employees in Austin, TX and Redwood City, CA
- \$91M in venture funding:
 - Softbank, DFJ, Lux, Valor Equity Partners, Lockheed Martin, Micron and others



Mike Henry-
Redwood City, CA

FOUNDER, CEO



Dave Fick-
Austin, TX

FOUNDER, CTO



Outline

1. AI inference at the Edge
2. Analog Compute-in-Memory
3. Mythic IPU Dataflow Architecture
4. Towards using OpenMP to program Mythic IPUs



— AI inference at the edge

Neural Networks = *Intuition*

Classification



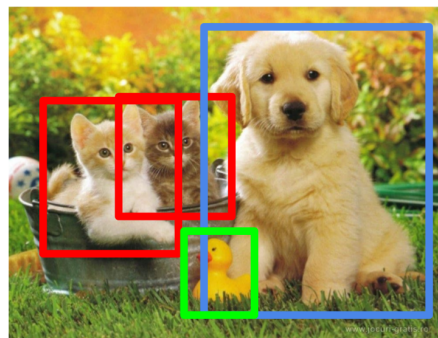
CAT

**Classification
+ Localization**



CAT

Object Detection



CAT, DOG, DUCK

**Instance
Segmentation**



CAT, DOG, DUCK

Single object


Multiple objects



Deep Neural Networks (DNNs) are Dominated by Multiply-Accumulate (MAC) Operations

Primary DNN Calculation is Input Vector * Weight Matrix = Output Vector

Input Data	Neuron Weights	Outputs Equations
$[X_0 \quad X_1 \quad \dots \quad X_N]$	$\begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ \dots & \dots & \dots \\ A_N & B_N & C_N \end{bmatrix}$	$= \begin{bmatrix} Y_A = X_0A_0 + X_1A_1 + X_2A_2 \\ Y_B = X_0B_0 + X_1B_1 + X_2B_2 \\ Y_C = X_0C_0 + X_1C_1 + X_2C_2 \end{bmatrix}$



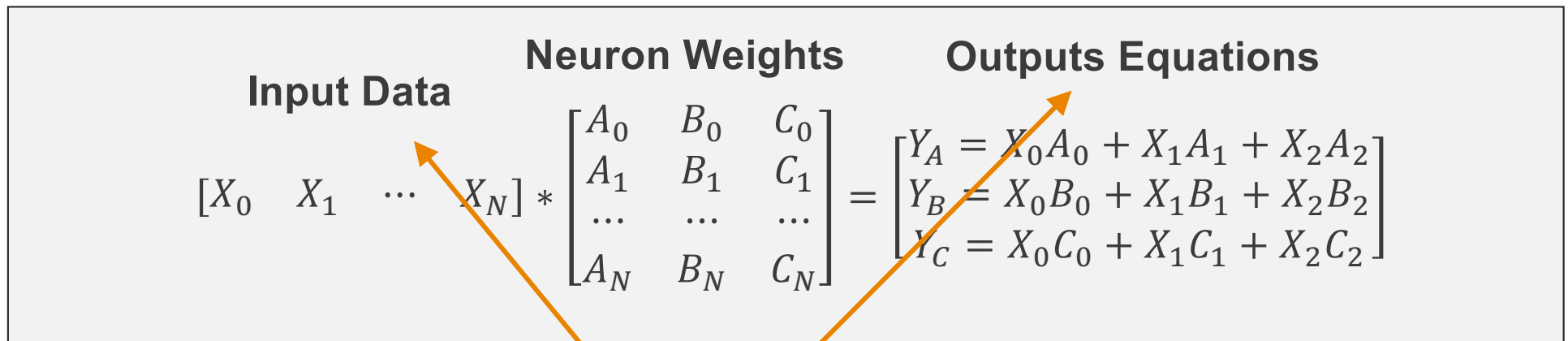
Key Operation: Multiply-Accumulate, or “MAC”

Figure of Merit: How many picojoules to execute a MAC?



Memory Access Includes Weight Data and Intermediate Data

“Weight Data”



“Intermediate Data”



Intermediate Data Accesses are Naturally Reused

For a 1000 input, 1000 neuron matrix....

$$\begin{array}{ccc} \text{1,000 Inputs} & \text{1,000,000 Weights} & \text{1,000 Outputs} \\ [X_0] \text{ } X_1 \text{ } \dots \text{ } X_N & * \begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ \dots & \dots & \dots \\ A_N & B_N & C_N \end{bmatrix} & = \begin{bmatrix} Y_A = X_0 A_0 + X_1 A_1 + X_2 A_2 \\ Y_B = X_0 B_0 + X_1 B_1 + X_2 B_2 \\ Y_C = X_0 C_0 + X_1 C_1 + X_2 C_2 \end{bmatrix} \end{array}$$

Intermediate data accesses are amortized **64-1024x**
since they are used in many MAC operations



Weight Data Accesses are Not Reused

For a 1000 input, 1000 neuron matrix....

$$\begin{array}{ccc} \text{1,000 Inputs} & \text{1,000,000 Weights} & \text{1,000 Outputs} \\ [X_0 \quad X_1 \quad \dots \quad X_N] * \begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ \dots & \dots & \dots \\ A_N & B_N & C_N \end{bmatrix} & = & \begin{bmatrix} Y_A = X_0A_0 + X_1A_1 + X_2A_2 \\ Y_B = X_0B_0 + X_1B_1 + X_2B_2 \\ Y_C = X_0C_0 + X_1C_1 + X_2C_2 \end{bmatrix} \end{array}$$

Weight data may need to be stored in *DRAM*, and it does not have the same amortization as the intermediate data



DNN Processing is All About Weight Memory

- 10+M parameters to store
- 20+B memory accesses
- How do we achieve...
 - High Energy Efficiency
 - High Performance
 - “Edge” Power Budget (e.g., 5W)

Network	Weights	MACs	...@ 30 FPS
AlexNet ¹	61 M	725 M	22 B
ResNet-18	11 M	1.8 B	54 B
ResNet-50	23 M	3.5 B	105 B
VGG-19 ¹	144 M	22 B	660 B
OpenPose ²	46 M	180 B	5400 B

**Very hard to fit this
in an Edge solution**

¹: 224x224 resolution
²: 656x368 resolution



Common Techniques for Reducing Weight Energy Consumption

■ Weight Re-use

■ Focus on CNN

- Re-use weights for multiple windows
- Can build specialized structures
- ☹ *Not all problems map to CNN well*

■ Focus on Large Batch

- Re-use weights for multiple inputs
- ☹ *Edge is often batch=1*
- ☹ *Increases latency*

■ Weight Reduction

■ Shrink the Model

- Use a smaller network that can fit on-chip (e.g., SqueezeNet)
- ☹ *Possibly reduced capability*

■ Compress the Model

- Use sparsity to eliminate up to 99% of the parameters
- Use literal compression
- ☹ *Possibly reduced capability*

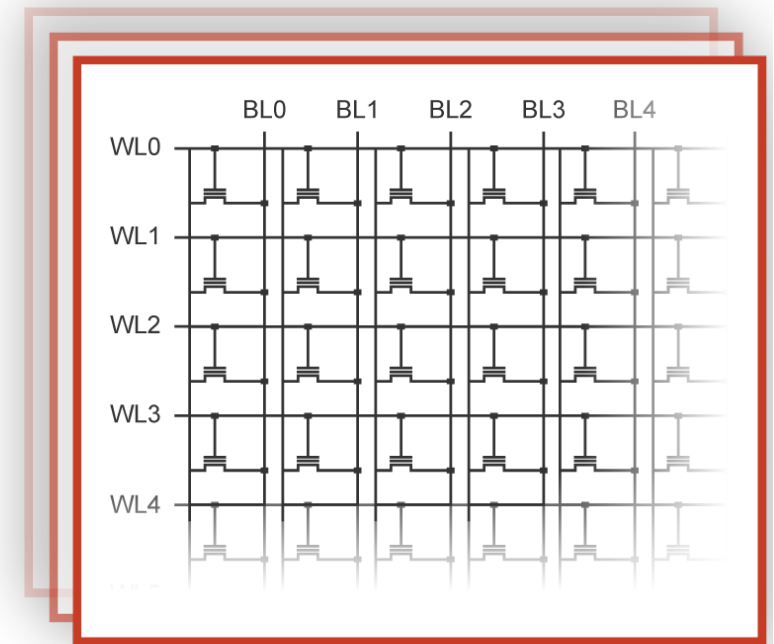
■ Reduce Weight Precision

- 32b Floating Point => 2-8b Integer
- ☹ *Possibly reduced capability*



Mythic's Matrix Multiplying Memory

- ✓ Never read weights
- ✓ This effectively makes weight memory access **energy-free** (only pay for MAC)
- ✓ And eliminates the need for...
 - Batch > 1
 - CNN Focus
 - Sparsity or Compression
 - Nerfed DNN Models



*Made possible with
Mixed-Signal Computing
on embedded flash*



Common NN Accelerator Design Points

Mythic is Fundamentally different

	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 is Fundamentally Different

	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				None
Precision	32f / 16f 8i	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

Also, Mythic does this in a 40nm process, compared to 7/10/16nm



— Analog Compute in Memory



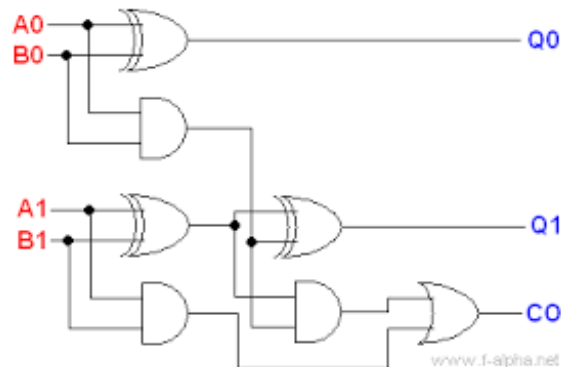
What Does “Analog Compute” Mean?

A type of computer that uses the continuously changeable aspects of physical phenomena such as *electrical* quantities to model the problem being solved. In contrast, **digital computers** represent varying quantities symbolically.

Problem: 2+2

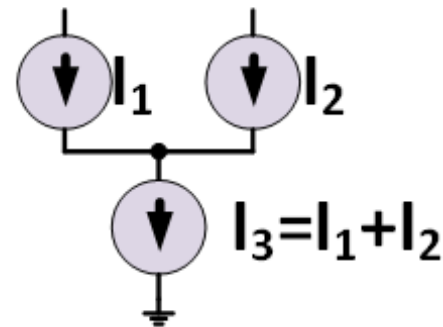
Digital Solution

$$0b10 + 0b10 = 0b100$$



A0 = 0
A1 = 1
B0 = 0
B1 = 1

Analog Solution

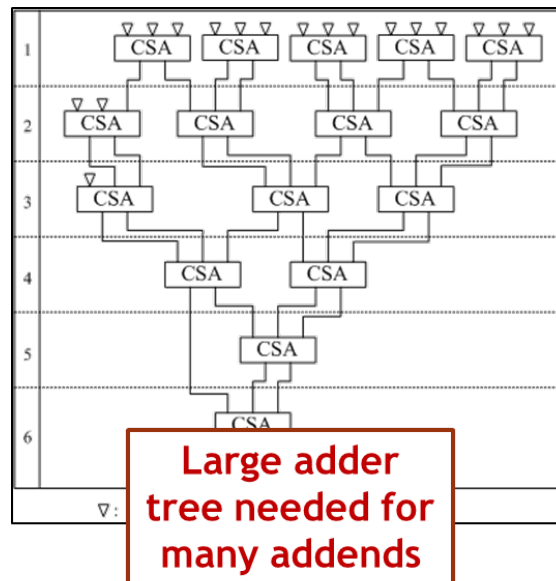


$I_1 = 2\mu A$
 $I_2 = 2\mu A$
 $I_3 = 4\mu A$

Why and When is Analog Compute Useful?

Digital Compute:

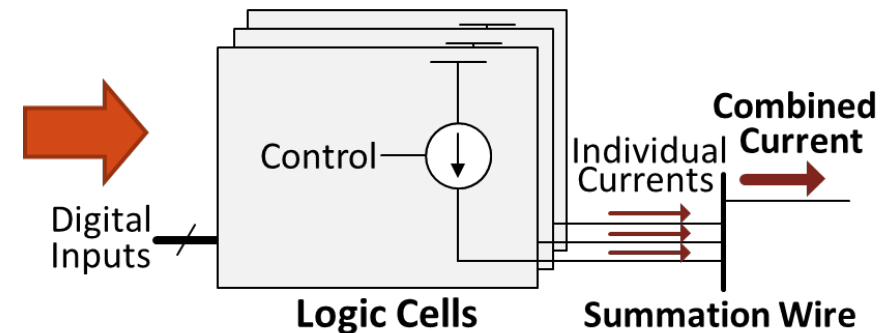
8168 full adders, 15 stage tree



1. Large problems
2. Noise tolerant algorithms

Analog Compute:

Current-mode summation, Single summation wire





Revisiting Matrix Multiply

Primary DNN Calculation is Input Vector * Weight Matrix = Output Vector

Input Data	Neuron Weights	Outputs Equations
$[X_0 \quad X_1 \quad \dots \quad X_N]$	$\begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ \dots & \dots & \dots \\ A_N & B_N & C_N \end{bmatrix}$	$\begin{bmatrix} Y_A = X_0A_0 + X_1A_1 + X_2A_2 \\ Y_B = X_0B_0 + X_1B_1 + X_2B_2 \\ Y_C = X_0C_0 + X_1C_1 + X_2C_2 \end{bmatrix}$

Flash Transistors



Analog Circuits Implement the MAC Operation

Flash transistors can be modeled as **variable resistors** representing the weight

The $V=IR$ current equation achieves the math we need:

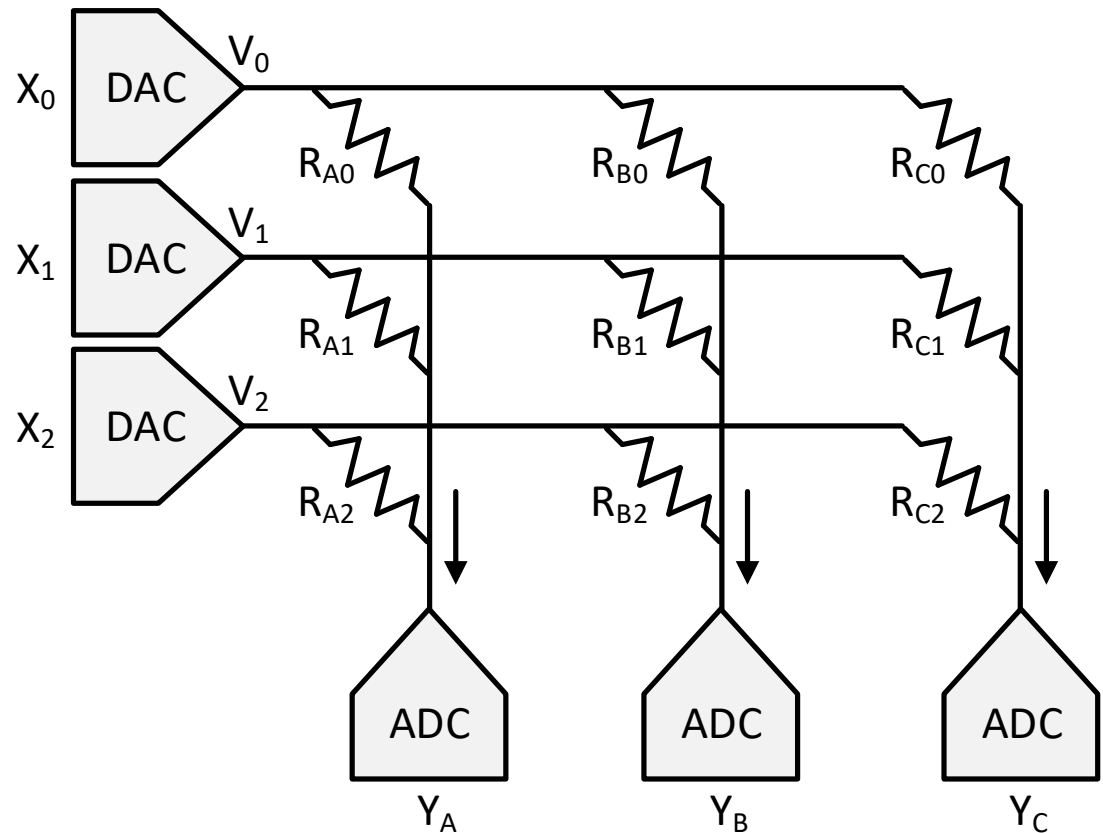
Inputs (X) = DAC

Weights (R) = Flash transistors

Outputs (Y) = ADC Outputs

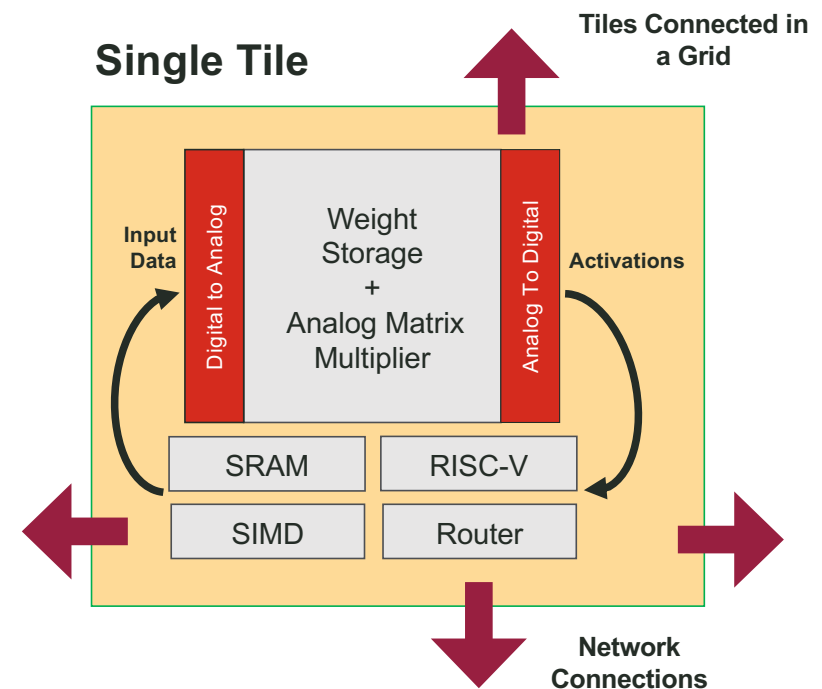
The ADCs convert current to digital codes, and provide the non-linearity needed for DNN

Eliminating weight movement and using analog computation provides >10x **overall** efficiency improvement vs digital systems

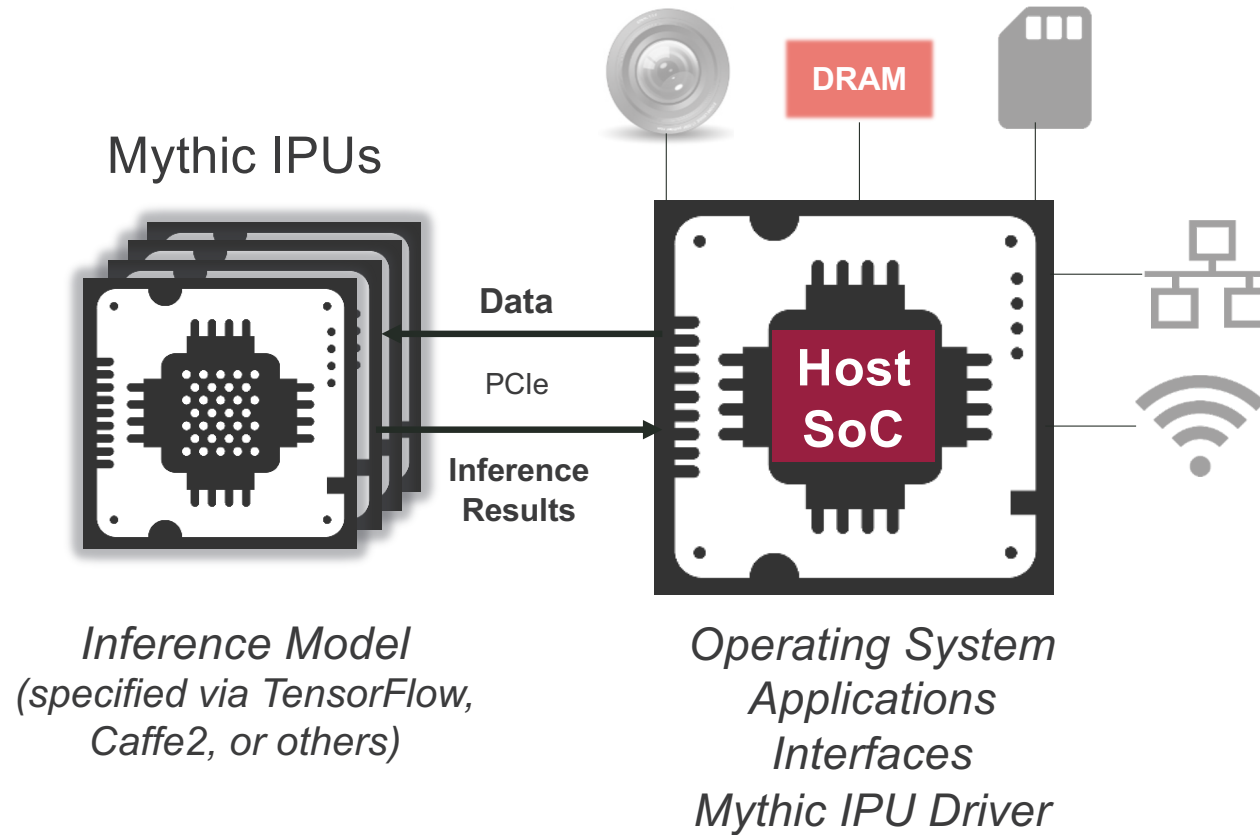


Mythic Mixed Signal Computing

- Downsides of Analog Computation
 - Noise! → compute using *changing* signals introduces noise
 - Flexibility
- Mixed Signal
 - Use analog where analog is best and digital where digital is best

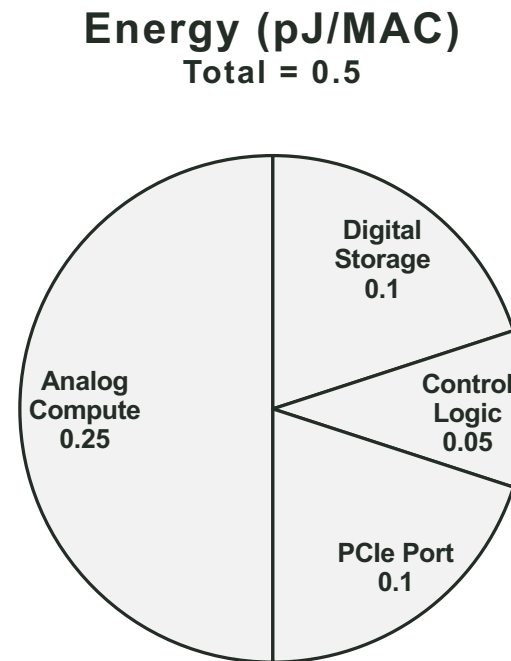


Mythic IPU is a PCIe Accelerator

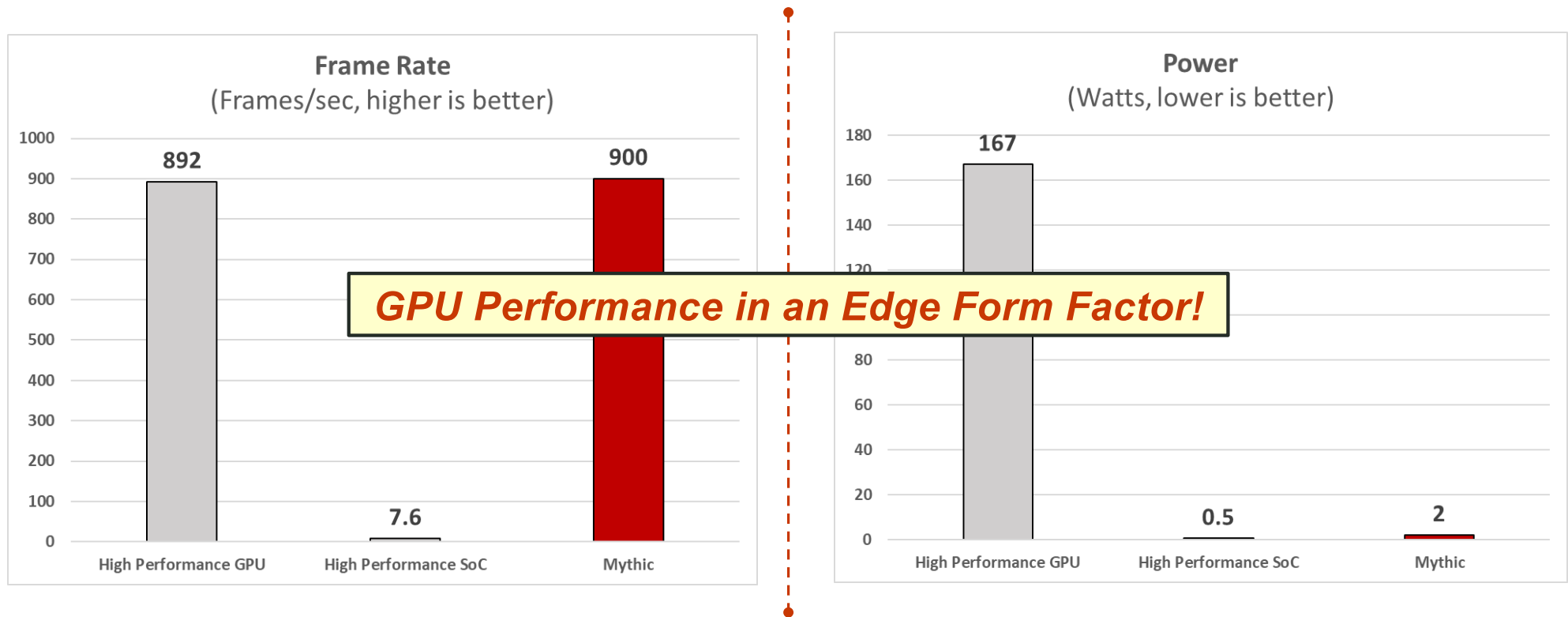


Energy consumption

- Numbers are for a typical application, e.g. ResNet-50
 - Batch size = 1
 - We are relatively application-agnostic (especially compared to DRAM-based systems)
- 8b analog compute accounts for about half of our energy
 - We can also run lower precision
 - Control, storage, and PCIe accounts for the other half



Example Application: ResNet-50



Running at 224x224 resolution. Mythic estimated, GPU/SoC measured



— Dataflow Architecture



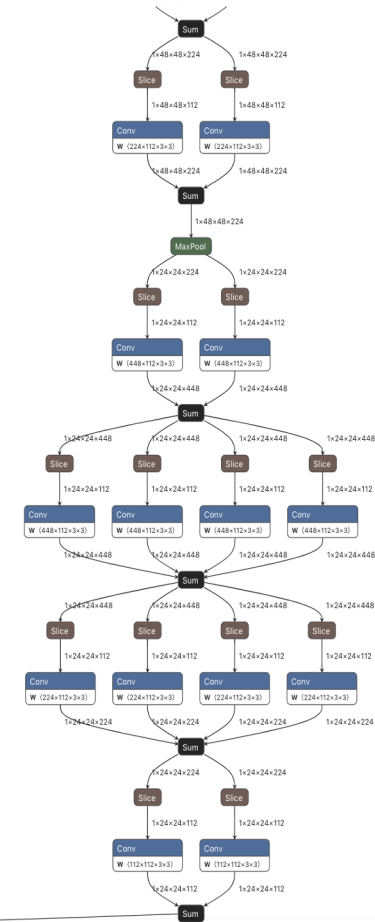
Neural Networks are Dataflow Graphs

Each operation depends on input data

- Understood as producer / consumer relationships

Many opportunities for parallelism,

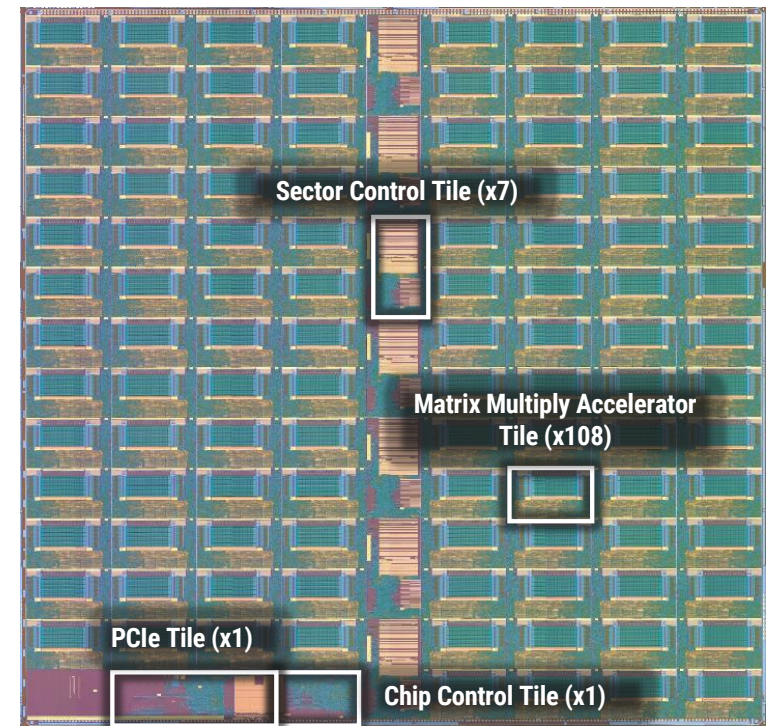
- but with many dependencies to manage





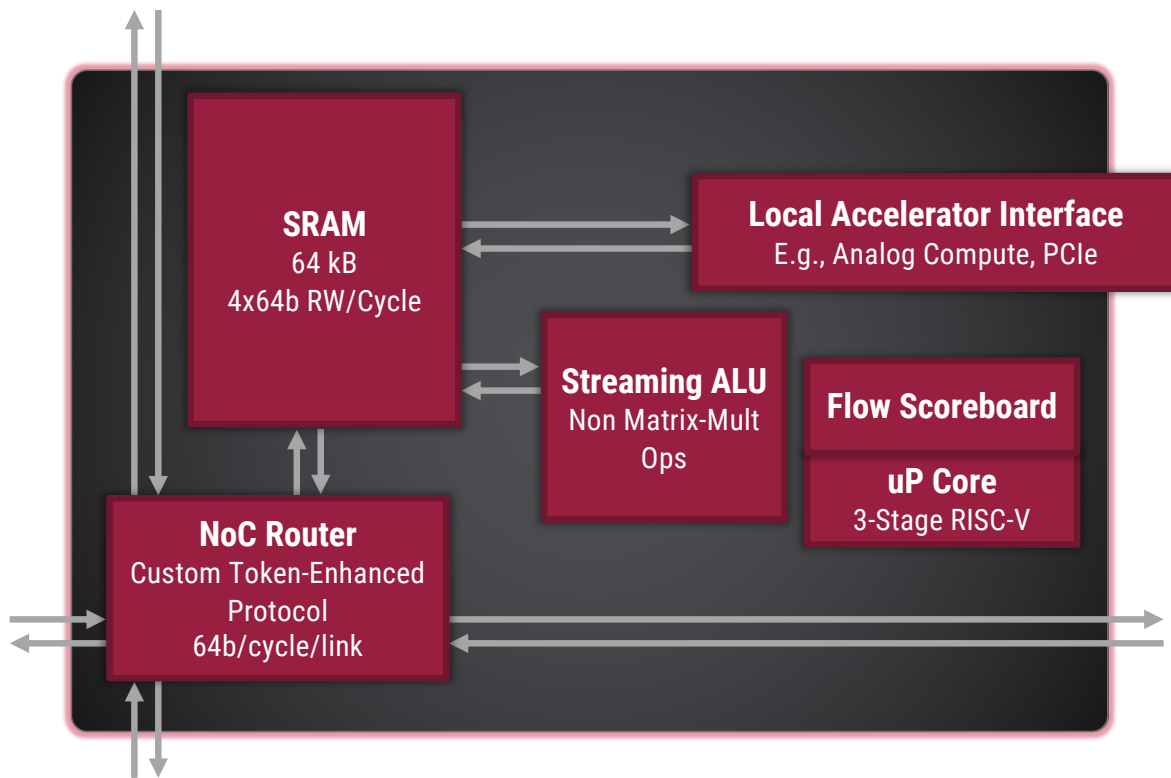
Mythic Uses a Graph-Based Dataflow Architecture

- Software overhead is eliminated by **automatically** starting operations when prerequisites are met
- Made possible by having producer / consumer relationships as a first-class architectural concept
- Matrix Multiply Accelerators (MMAs) operations are nodes with intermediate data flowing between
- Communication and synchronization is handled by sending tokens and updating a 2-level Flow Scoreboard (FSB)





Mythic Tile Foundation

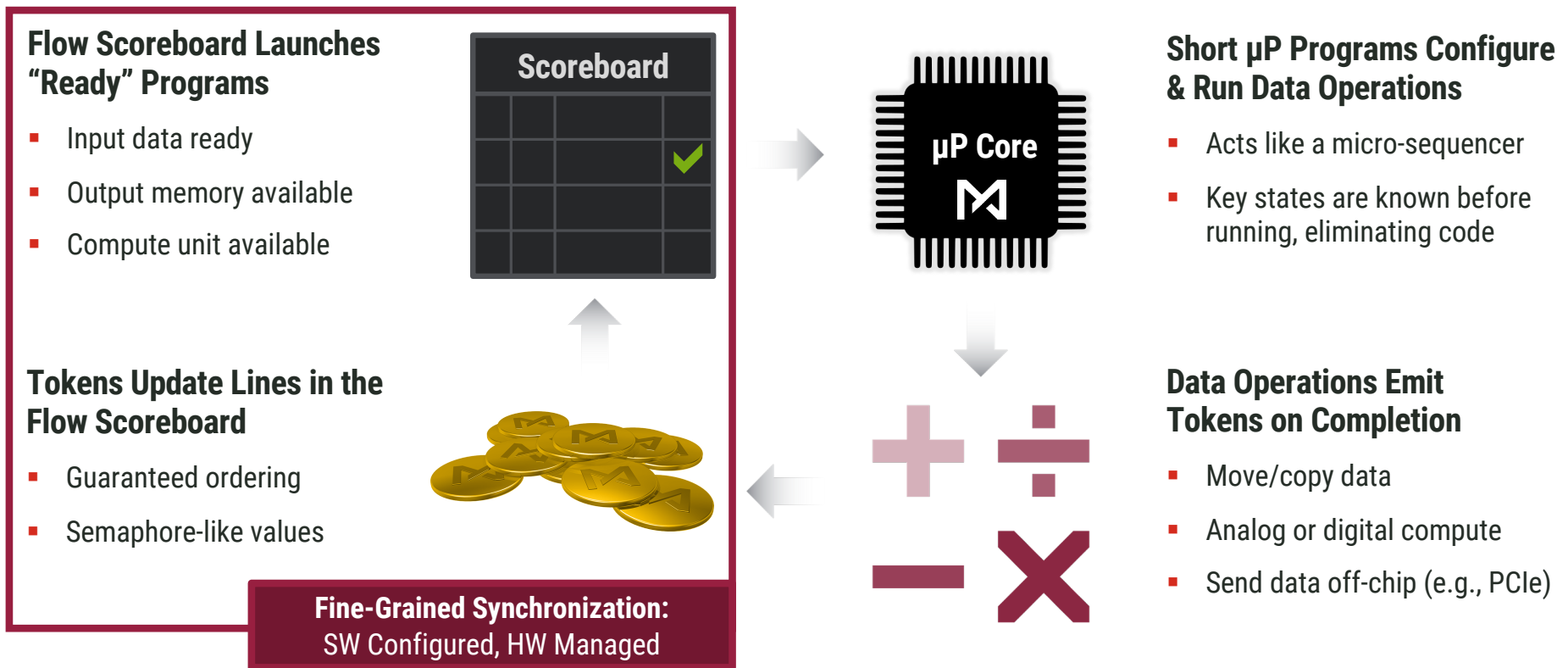


- Flow Scoreboard (FSB)
- Microprocessor (uP) Core
- Memory
- Streaming ALU (simd)
- Network-on-Chip (NoC)
- Local Accelerator Interface

... consistency helps the compiler!

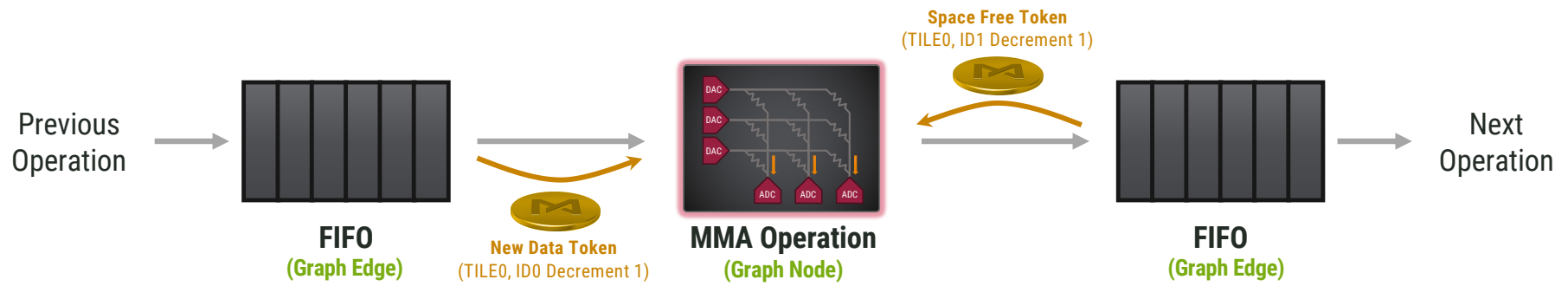


Dataflow is Managed via Tokens and the Flow Scoreboard





Flexible Two-Level Flow Scoreboard



Token Table

Tracks Each Dependency With Conditions

ID	Count	Cond.	Prog +/-	Prog ID
0	9	≤ 0	Minus	0
1	-8	< 0	Minus	0
2	Expandable to support multiple producers and consumers.			
...				

Program Table

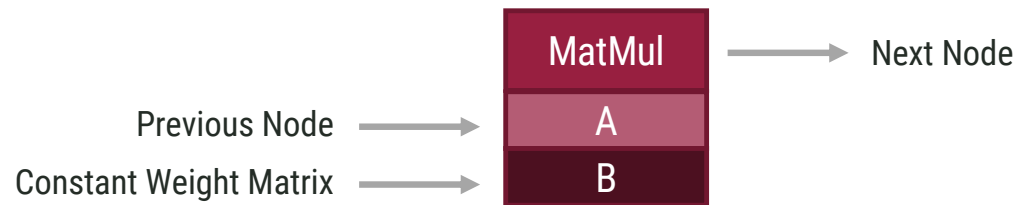
Aggregates Dependencies With Conditions

ID	Count	Cond.	Unit Mask	Prog Ptr
0	2	$= 0$	MMA	0xABCD
1				
2				
...				

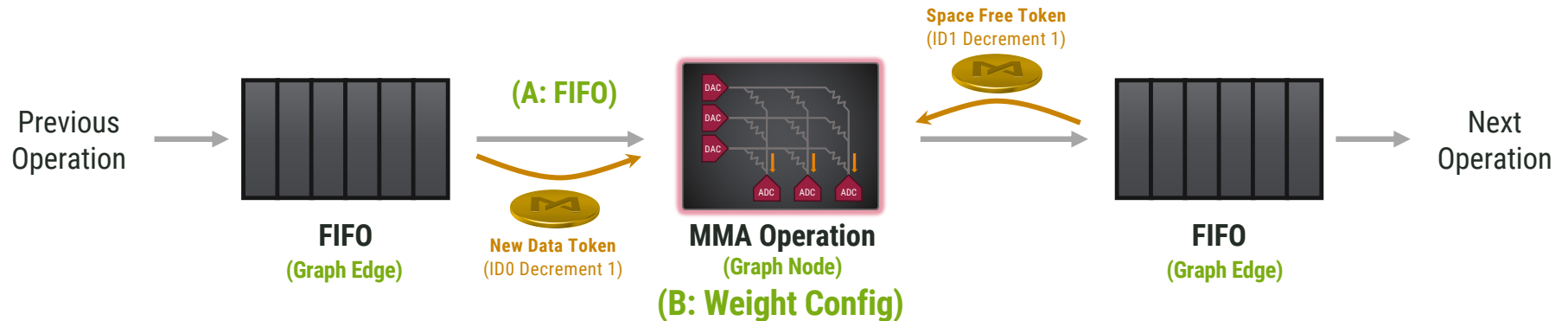


Graph Compiler Handles Flow Scoreboard Configuration

Open Neural Network Exchange (ONNX)

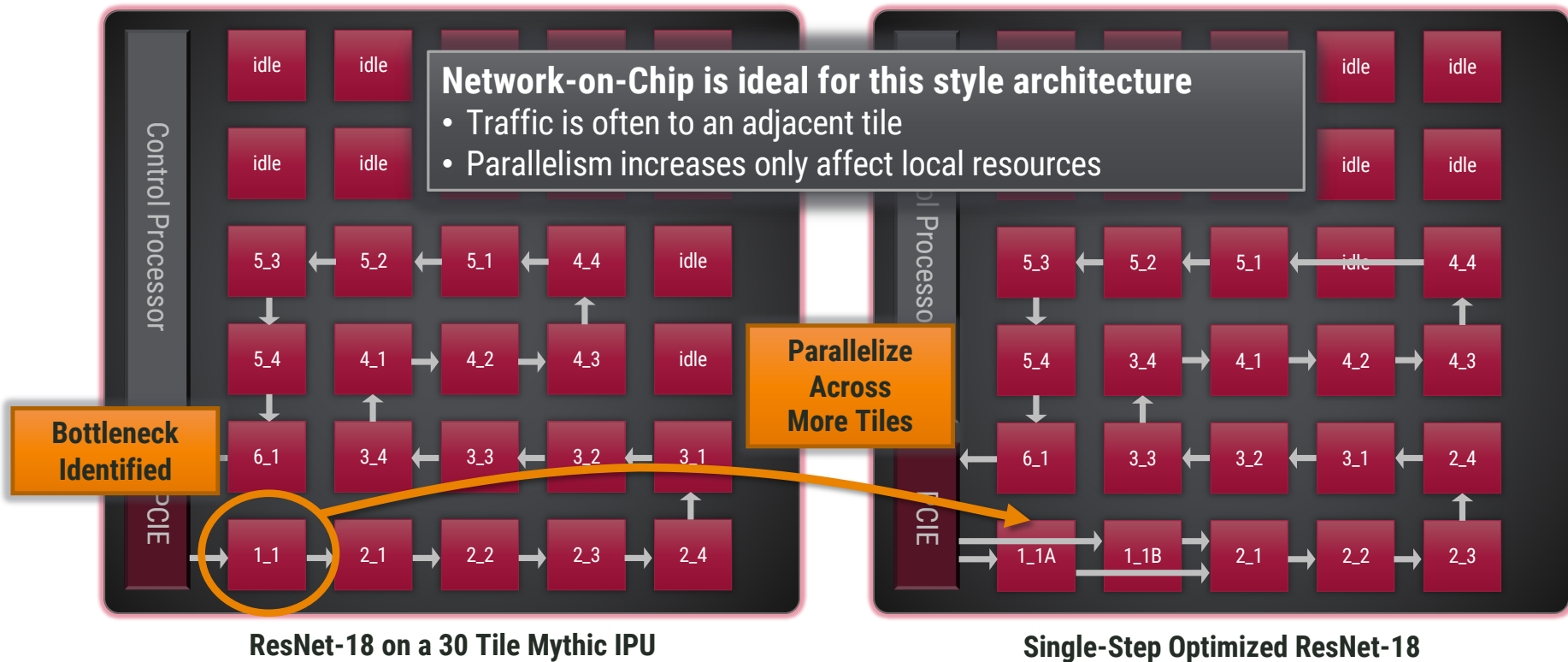


Each graph element has a Mythic Flow Scoreboard translation





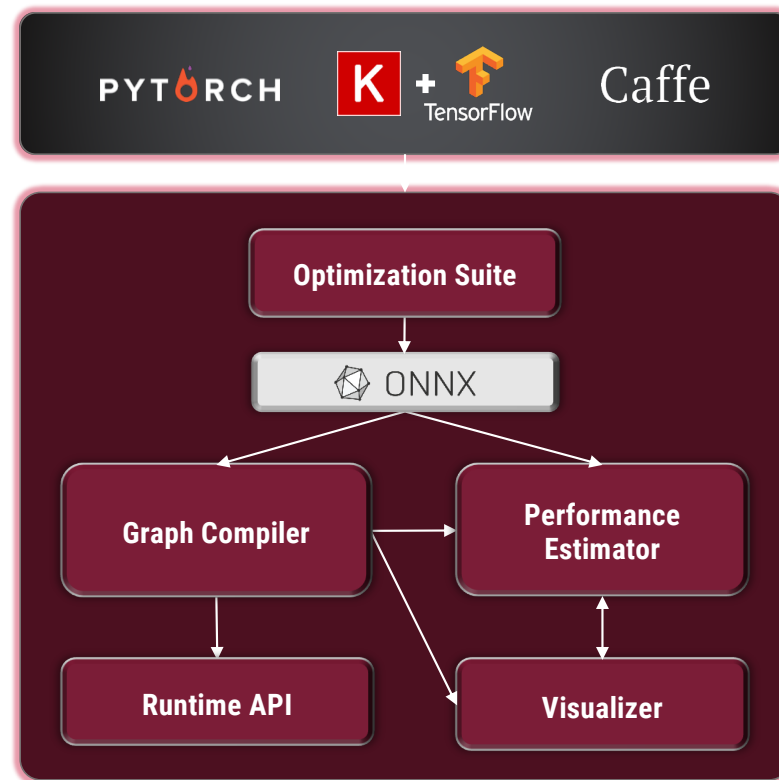
Graph Compiler Optimizes The Graph Throughput





Mythic SDK Enables Developers With The Latest Frameworks

- Graph decomposition and mapping
- Code generation
- Host runtime API and OS drivers



- Post-training quantization
- Retraining libraries
- Annotated
- Power, speed and memory estimates
- Profiling and logging

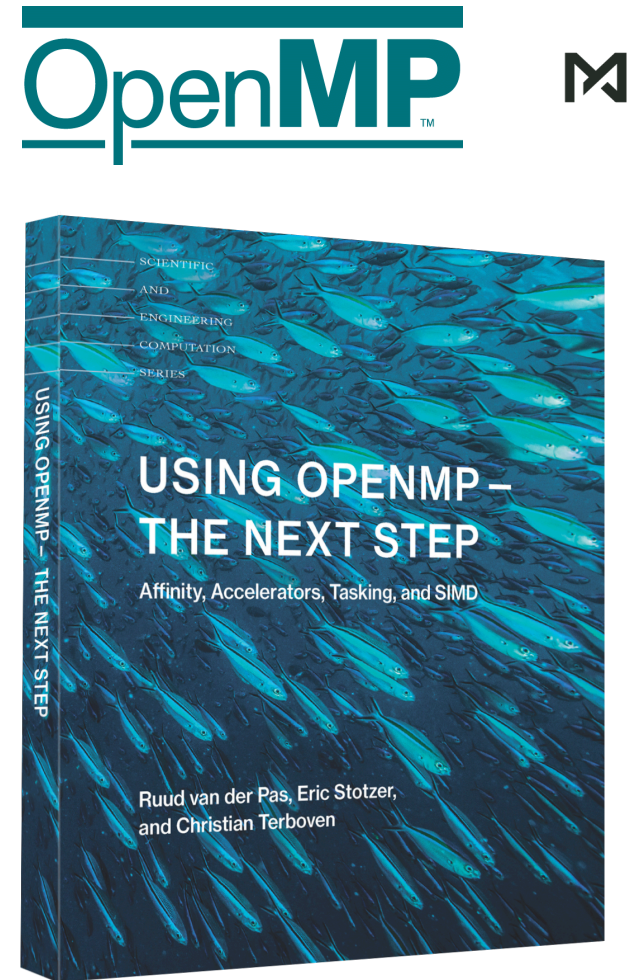


Towards using OpenMP to program Mythic IPU's

Caveat – These are the presenter's ideas and do not reflect any Mythic plans

Acknowledgements

- Material borrowed from SC18 Tutorials
- Programming Your GPU with OpenMP
 - Tim Mattson (Intel), Simon McIntosh-Smith (U. of Bristol), Eric Stotzer
- Mastering Tasking with OpenMP
 - Christian Terboven (RWTH Aachen), Michael Klemm (Intel), Sergi Mateo Bellido (BSC), Xavier Teruel (BSC), and Bronis R. de Supinski (LLNL)
- To learn more about programming with OpenMP, get a copy of this awesome book!
<https://mitpress.mit.edu/books/using-openmp-next-step>



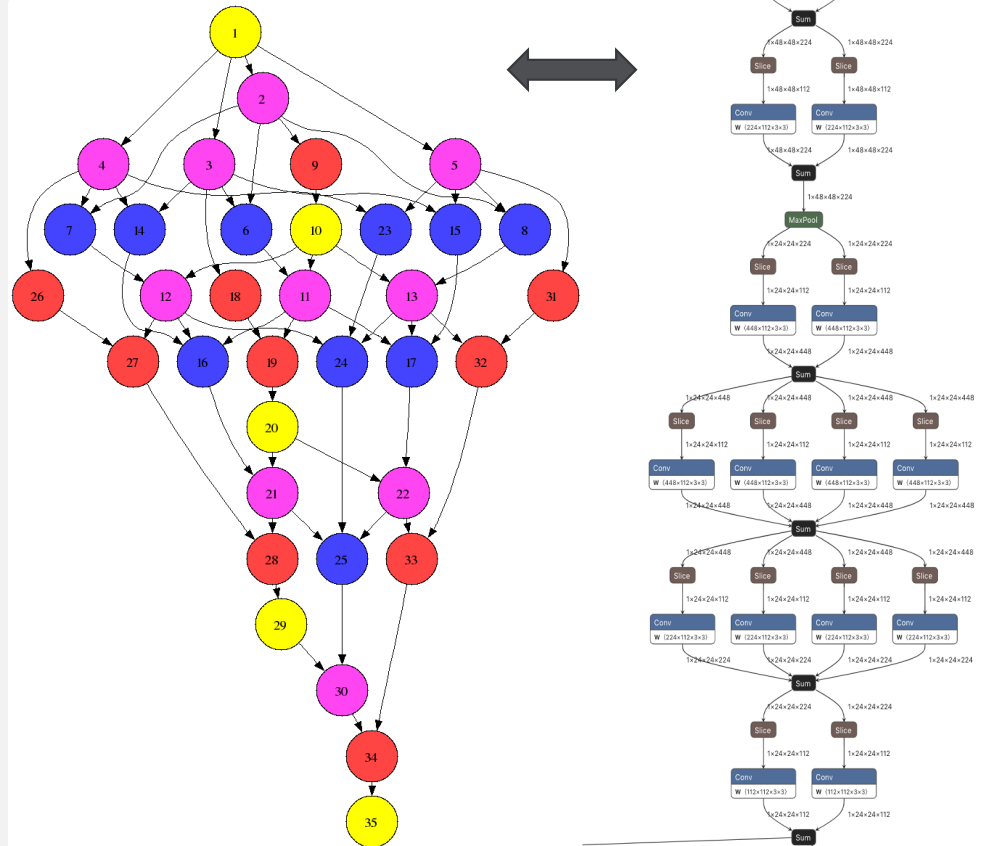
OpenMP Task Dependence and DNN Data Flow Models

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        // Diagonal Block factorization
        #pragma omp task depend(inout: a[k][k])
        potrf(a[k][k], ts, ts);

        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task depend(in: a[k][k])
                           depend(inout: a[k][i])
            trsm(a[k][k], a[k][i], ts, ts);
        }

        // Update trailing matrix
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
                #pragma omp task depend(inout: a[j][i])
                               depend(in: a[k][i], a[k][j])
                dgermm(a[k][i], a[k][j], a[j][i], ts, ts);
            }
            #pragma omp task depend(inout: a[i][i])
                           depend(in: a[k][i])
            syrkm(a[k][i], a[i][i], ts, ts);
        }
    }
}
```

OpenMP 4.0



DNN Graphs are a Domain Specific Language (DSL)

```
graph AlexNet( input ) -> ( output )
{
  input = external(shape = [1, 3, 224, 224]);
  kernel1 = variable(shape = [64, 3, 11, 11], label = 'alexnet_v2/conv1/kernel');
  bias1 = variable(shape = [1, 64], label = 'alexnet_v2/conv1/bias');
  #pragma omp loop block
  conv1 = conv(input, kernel1, bias1, padding = [(0,0), (0,0)],
  border = 'constant', stride = [4, 4], dilation = [1, 1]);
  relu1 = relu(conv1);
  pool1 = max_pool(relu1, size = [1, 1, 3, 3], stride = [1, 1, 2, 2],
  border = 'ignore', padding = [(0,0), (0,0), (0,0), (0,0)]);
  kernel2 = variable(shape = [192, 64, 5, 5], label = 'alexnet_v2/conv2/kernel');
  bias2 = variable(shape = [1, 192], label = 'alexnet_v2/conv2/bias');
  #pragma omp loop diag
  conv2 = conv(pool1, kernel2, bias2, padding = [(2,2), (2,2)],
  border = 'constant', stride = [1, 1], dilation = [1, 1]);
  relu2 = relu(conv2);
  pool2 = max_pool(relu2, size = [1, 1, 3, 3], stride = [1, 1, 2, 2],
  border = 'ignore', padding = [(0,0), (0,0), (0,0), (0,0)]);
  kernel3 = variable(shape = [384, 192, 3, 3], label = 'alexnet_v2/conv3/kernel');
  bias3 = variable(shape = [1, 384], label = 'alexnet_v2/conv3/bias');
  conv3 = conv(pool2, kernel3, bias3, padding = [(1,1), (1,1)],
  border = 'constant', stride = [1, 1], dilation = [1, 1]);
  relu3 = relu(conv3);
}
```

- A subset of a general-purpose language
- Regular Looping patterns
- No pointers and memory aliasing etc...
- Each operation is a function that operates on tensors



Conv2d – Convolution Layer Example

```
#define I 112 // Input channels
#define O 112 // Output channels
#define F 3 // Filter
#define K 7 // Domain

int8_t src[K+2][K+2][I]; // 1*9*9*112
int16_t dst[K][K][O]; // 1*7*7*112
int8_t W[F][F][I][O]; // (3*3*112)*112
int8_t Wt[O][F][F][I]; // Tranpose of W 112*(3*3*112)

void conv2d_inner_channels_loop()
{
    for (int k0=0; k0<7; k0++)
        for (int k1=0; k1<7; k1++)
            for (int oc=0; oc<112; oc++)
                for (int ic=0; ic<112; ic++)
                    for (int f0=0; f0<3; f0++)
                        for (int f1=0; f1<3; f1++)
                            /* 1x1x128 += 1x[1x1x112] * [1x1x112]x112 */
                            dst[k0][k1][oc] += src[k0+f0][k1+f1][ic] * Wt[oc][f0][f1][ic][oc]; // Use MMADOT here.
}
```



Conv2d with Loop Unroll and Array Sections

```
/* Imagine a world with array sections in C. */

void conv2d_unroll_output_input_channels_loop()
{
    for (int k0=0; k0<7; k0++)
        for (int k1=0; k1<7; k1++)
            // for (int oc=0; ocd<112; oc++)
            // for (int ic=0; icd<112; ic++)
            for (int f0=0; f0<3; f0++)
                for (int f1=0; f1<3; f1++)
                    /* 1x1x128 += 1x[1x1x112] * [1x1x112]x112 */
                    dst[k0][k1][:112] += src[k0+f0][k1+f1][:112] * W[f0][f1][:112][:112];
}

void conv2d_unroll_filter_loops()
{
    for (int k0=0; k0<7; k0++)
        for (int k1=0; k1<7; k1++)
            // for (int oc=0; ocd<112; oc++)
            // for (int ic=0; icd<112; ic++)
            // for (int f0=0; f0<3; f0++)
            // for (int f1=0; f1<3; f1++)
            /* 1x1x128 += [1x1]x[3x3x112] * [3x3x112]x112 */
            dst[k0][k1][:112] += src[k0:3][k1:3][:112] * W[:3][:3][:112][:112]; // MYTHIC_MMADOT()
}
```

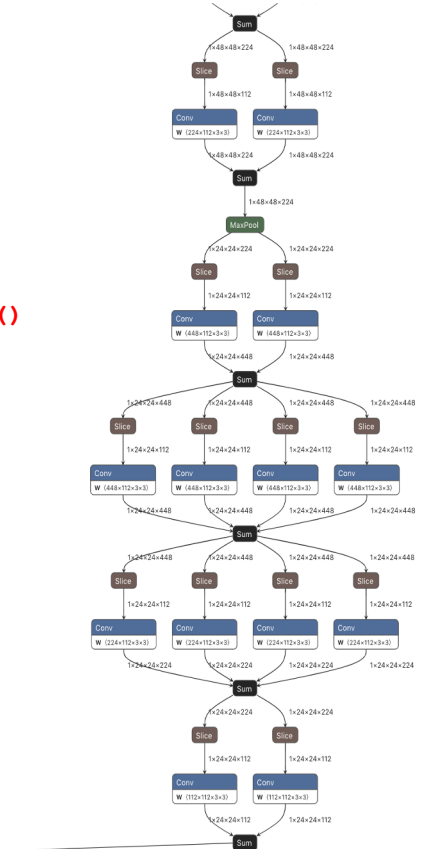


Leverage FSB for Data Flow across Tasks assigned to tiles

```
void conv2d_mythic_mmaddot()
{
    for (int k0=0; k0<7; k0++)
        for (int k1=0; k1<7; k1++)
        {
            /* SRC_FSB >= f(k0,k1)*112 && DST_FSB < g(k0,k1)*112) { SRC_FSB -= 112, DST_FSB += 112 */
            #pragma omp task depend(IN(src[k0:3][k1:3][:112]) \
                                   depend(OUT(dst[k0][k1][:112])))
            dst[k0][k1][:112] += src[k0:3][k1:3][:112] * W[:3][:3][:112][:112]; // MYTHIC_MMADOT()

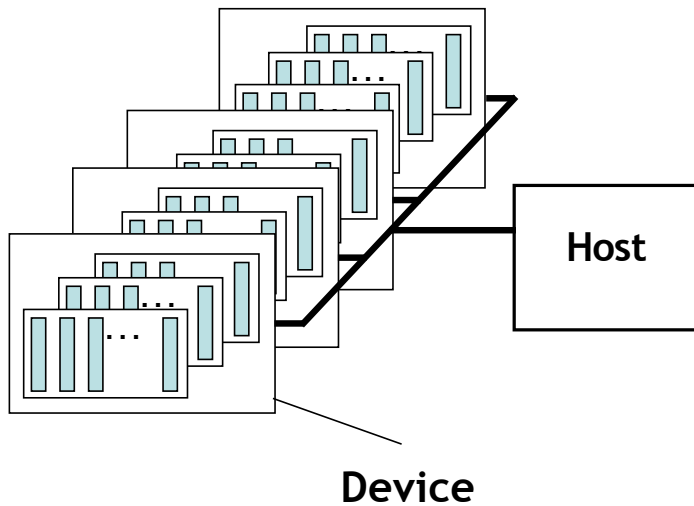
            /* REMOTE_SRC_FSB -= f(k0,k1)*112, REMOTE_DST_FSB += g(k0,k1)*112 */
        }
    }
}
```

- W (Weights) are constant.
- Src (input) flows from a previous layer (task) in the DNN graph.
- Dst (output) flows to a successor layer (task) in the DNN graph.
- Task dependence is implemented by the Mythic IPU Flow Scoreboard (FSB)
- Updates to FSB are via f() and g() which are functions of the src and dst buffer sizes and respective iterations.



Heterogeneous Accelerators

- OpenMP uses a host/device model
 - The host is where the initial thread of the program begins execution
 - Zero or more devices are connected to the host
 - Device-memory address space is distinct from host-memory address space



```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
           omp_get_num_devices());
}
```



Use OpenMP accelerator model to offload to Mythic IPU?

- A Mythic IPU is a microcosm of a super-computer
 - Each Tile is a Node, distributed memory, message passing inter-tile
 - Tile/Node shared memory, OpenMP+Accelerator model intra-tile
- Mythic-IPU Accelerator model
 - The PCIE tile is the master tile that offloads work to other tiles
 - Every task is offloaded to a tile identified by its x,y rank?
- Host+Mythic-IPU accelerator model
 - A host processor is the master device that offloads work to tiles on one or more IPU.

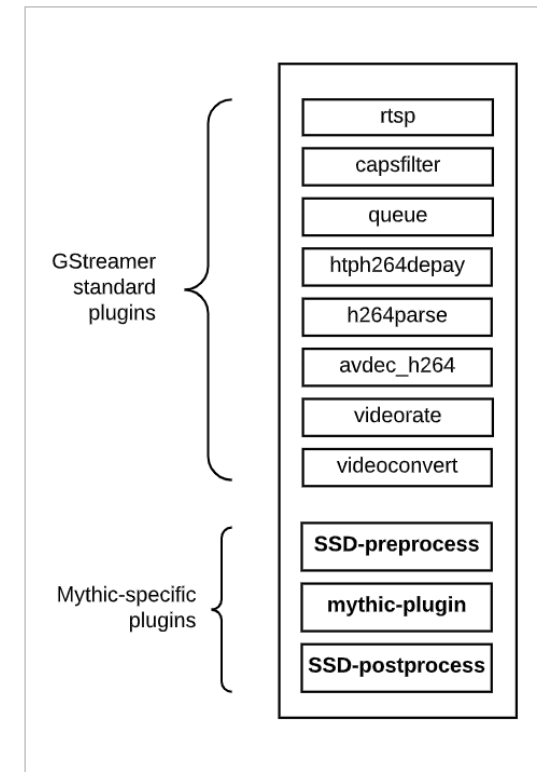
OpenMP Task/Target with depend

```
#pragma omp task depend(out:frame)
SDS_PreProcess(frame);

#pragma omp target device(ipu)
depend(inout:frame)
{
    DNN_Inference(frame);
}

#pragma omp task in(frame)
SDS_PostProcess(frame);

#pragma omp taskwait
```



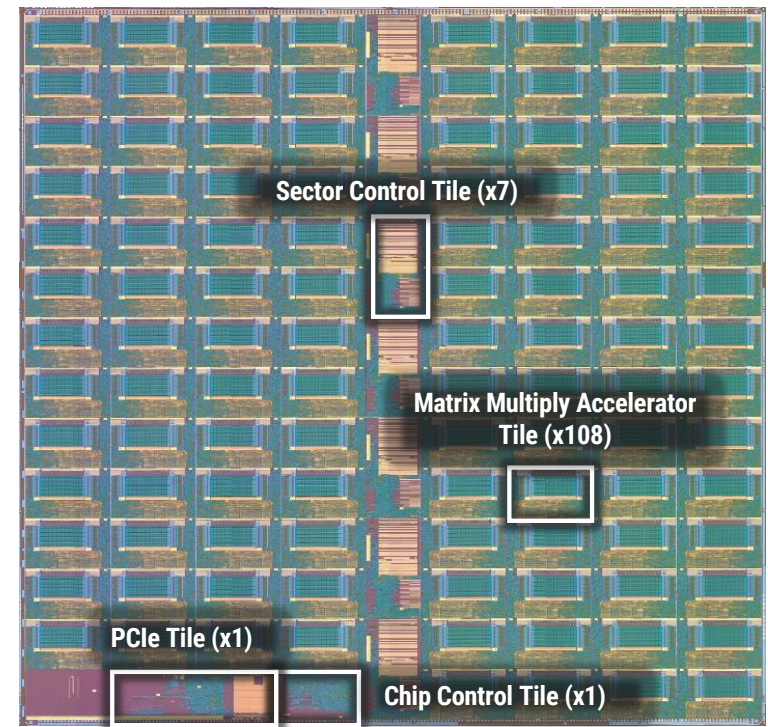


— Summary



Mythic IPU Overview

- ✓ Low Latency
 - Runs batch size = 1, single frame latency
- ✓ High Performance
 - 10's of TMAC/s
- ✓ High Efficiency
 - 0.5 pJ/MAC aka 500mW / TMAC
- ✓ Hyper-Scalable
 - Ultra low power to high performance
- ✓ Easy to use
 - Topology agnostic (CNN/DNN/RNN)
 - TensorFlow/Caffe2/etc supported





Summary

- Deep Neural Networks are dominated by MAC operations and are tolerant to noise and loss precision loss effects.
- Analog compute-in-memory provides for efficient matrix multiplication on AI inference applications.
- Mythic's IPU is a mixed-signal (digital+analog) AI inference accelerator.
- DNNs are really task graphs.
- Perhaps OpenMP Tasking and offload models could be used to program Mythic's IPU dataflow architecture.