Programming Models for a Mixed-Signal AI Inference Accelerator
IWOMP 2020

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
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**
  - Single object: CAT

- **Classification + Localization**
  - Single object: CAT

- **Object Detection**
  - Multiple objects: CAT, DOG, DUCK

- **Instance Segmentation**
  - Multiple objects: CAT, DOG, DUCK

© 2020 MYTHIC. ALL RIGHTS RESERVED.
Deep Neural Networks (DNNs) are Dominated by Multiply-Accumulate (MAC) Operations

Primary DNN Calculation is  \( \text{Input Vector} \times \text{Weight Matrix} = \text{Output Vector} \)

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Neuron Weights</th>
<th>Outputs Equations</th>
</tr>
</thead>
</table>
| \([X_0 \ X_1 \ \cdots \ X_N]\) | \[
\begin{bmatrix}
A_0 & B_0 & C_0 \\
A_1 & B_1 & C_1 \\
\vdots & \vdots & \vdots \\
A_N & B_N & C_N
\end{bmatrix}
\] | \[
\begin{align*}
Y_A &= X_0A_0 + X_1A_1 + X_2A_2 \\
Y_B &= X_0B_0 + X_1B_1 + X_2B_2 \\
Y_C &= X_0C_0 + X_1C_1 + X_2C_2
\end{align*}
\] |

**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”

Input Data

\[
\begin{bmatrix}
    X_0 & X_1 & \ldots & X_N
\end{bmatrix}
\]

Neuron Weights

\[
\begin{bmatrix}
    A_0 & B_0 & C_0 \\
    A_1 & B_1 & C_1 \\
    \ldots & \ldots & \ldots \\
    A_N & B_N & C_N
\end{bmatrix}
\]

Outputs Equations

\[
\begin{align*}
    Y_A &= X_0A_0 + X_1A_1 + X_2A_2 \\
    Y_B &= X_0B_0 + X_1B_1 + X_2B_2 \\
    Y_C &= X_0C_0 + X_1C_1 + X_2C_2
\end{align*}
\]

“Intermediate Data”
Intermediate Data Accesses are Naturally Reused

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

\[
\begin{bmatrix}
X_0 \\
X_1 \\
\vdots \\
X_N
\end{bmatrix} \times \begin{bmatrix}
A_0 & B_0 & C_0 \\
A_1 & B_1 & C_1 \\
\vdots & \vdots & \vdots \\
A_N & B_N & C_N
\end{bmatrix} = \begin{bmatrix}
Y_A = X_0A_0 + X_1A_1 + X_2A_2 \\
Y_B = X_0B_0 + X_1B_1 + X_2B_2 \\
Y_C = X_0C_0 + X_1C_1 + X_2C_2
\end{bmatrix}
\]

Intermediate data accesses are amortized $64\text{-}1024x$ since they are used in many MAC operations.
Weight Data Accesses are Not Reused

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

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)

<table>
<thead>
<tr>
<th>Network</th>
<th>Weights</th>
<th>MACs</th>
<th>…@ 30 FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet(^1)</td>
<td>61 M</td>
<td>725 M</td>
<td>22 B</td>
</tr>
<tr>
<td>ResNet-18(^1)</td>
<td>11 M</td>
<td>1.8 B</td>
<td>54 B</td>
</tr>
<tr>
<td>ResNet-50(^1)</td>
<td>23 M</td>
<td>3.5 B</td>
<td>105 B</td>
</tr>
<tr>
<td>VGG-19(^1)</td>
<td>144 M</td>
<td>22 B</td>
<td>660 B</td>
</tr>
<tr>
<td>OpenPose(^2)</td>
<td>46 M</td>
<td>180 B</td>
<td>5400 B</td>
</tr>
</tbody>
</table>

Very hard to fit this in an Edge solution

\(^1\): 224x224 resolution
\(^2\): 656x368 resolution

© 2020 MYTHIC. ALL RIGHTS RESERVED.
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

<table>
<thead>
<tr>
<th></th>
<th>Enterprise With DRAM</th>
<th>Enterprise No-DRAM</th>
<th>Edge With DRAM</th>
<th>Edge No-DRAM</th>
<th>Mythic NVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>&lt;50 MB</td>
<td>100+ MB</td>
<td>&lt; 5 MB</td>
<td>&lt; 5 MB</td>
<td>&lt; 5 MB</td>
</tr>
<tr>
<td>DRAM</td>
<td>8+ GB</td>
<td>-</td>
<td>4-8 GB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Power</td>
<td>70+ W</td>
<td>70+ W</td>
<td>3-5 W</td>
<td>1-3 W</td>
<td>1-5 W</td>
</tr>
<tr>
<td>Sparsity</td>
<td>Light</td>
<td>Light</td>
<td>Moderate</td>
<td>Heavy</td>
<td>None</td>
</tr>
<tr>
<td>Precision</td>
<td>32f / 16f / 8i</td>
<td>32f / 16f / 8i</td>
<td>8i</td>
<td>1-8i</td>
<td>1-8i</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Great</td>
<td>Great</td>
<td>Moderate</td>
<td>Poor</td>
<td>Great</td>
</tr>
<tr>
<td>Performance</td>
<td>High</td>
<td>High</td>
<td>Very Low</td>
<td>Very Low</td>
<td>High</td>
</tr>
</tbody>
</table>
### Mythic is Fundamentally Different

<table>
<thead>
<tr>
<th></th>
<th>Enterprise With DRAM</th>
<th>Enterprise No-DRAM</th>
<th>Edge With DRAM</th>
<th>Edge No-DRAM</th>
<th>Mythic NVM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRAM</strong></td>
<td>&lt;50 MB</td>
<td>100+ MB</td>
<td>&lt; 5 MB</td>
<td>&lt; 5 MB</td>
<td>&lt; 5 MB</td>
</tr>
<tr>
<td><strong>DRAM</strong></td>
<td>8+ GB</td>
<td>-</td>
<td>4-8 GB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>70+ W</td>
<td>70+ W</td>
<td>3.5 W</td>
<td>1.3 W</td>
<td>1.5 W</td>
</tr>
<tr>
<td><strong>Sparsity</strong></td>
<td>Light</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>32f / 16f / 8i</td>
<td>8i</td>
<td>8i</td>
<td>1-8i</td>
<td></td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>Great</td>
<td>Great</td>
<td>Moderate</td>
<td>Poor</td>
<td>Great</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>High</td>
<td>High</td>
<td>Very Low</td>
<td>Very Low</td>
<td>High</td>
</tr>
</tbody>
</table>

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**

I1 = 2uA
I2 = 2uA
I3 = 4uA

\[ I_3 = I_1 + I_2 \]
Why and When is Analog Compute Useful?

Digital Compute:
- Large problems
- Noise tolerant algorithms

8168 full adders, 15 stage tree

Analog Compute:
- Current-mode summation, Single summation wire

Large adder tree needed for many addends

Single Wire Summation
Regardless of how many addends
Revisiting Matrix Multiply

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

\[
\begin{bmatrix} X_0 & X_1 & \cdots & X_N \end{bmatrix} \times \begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ \vdots & \vdots & \vdots \\ A_N & B_N & C_N \end{bmatrix} = \begin{bmatrix} Y_A = X_0A_0 + X_1A_1 + X_2A_2 \\ Y_B = X_0B_0 + X_1B_1 + X_2B_2 \\ Y_C = X_0C_0 + X_1C_1 + X_2C_2 \end{bmatrix}
\]

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

© 2020 MYTHIC. ALL RIGHTS RESERVED.
Mythic IPU is a PCIe Accelerator

Mythic IPUs

Inference Model
(specified via TensorFlow, Caffe2, or others)

Data
PCIe
Inference Results

Host
SoC

DRAM

Operating System
Applications
Interfaces
Mythic IPU Driver

22

© 2020 MYTHIC. ALL RIGHTS RESERVED.
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

**Flow Scoreboard Launches “Ready” Programs**
- Input data ready
- Output memory available
- Compute unit available

**Tokens Update Lines in the Flow Scoreboard**
- Guaranteed ordering
- Semaphore-like values

**Short μP Programs Configure & Run Data Operations**
- Acts like a micro-sequencer
- Key states are known before running, eliminating code

**Data Operations Emit Tokens on Completion**
- Move/copy data
- Analog or digital compute
- Send data off-chip (e.g., PCIe)
Flexible Two-Level Flow Scoreboard

Token Table
Tracks Each Dependency With Conditions

<table>
<thead>
<tr>
<th>ID</th>
<th>Count</th>
<th>Cond.</th>
<th>Prog +/-</th>
<th>Prog ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>&lt;= 0</td>
<td>Minus</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-8</td>
<td>&lt; 0</td>
<td>Minus</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Expandable to support multiple producers and consumers.

Program Table
Aggregates Dependencies With Conditions

<table>
<thead>
<tr>
<th>ID</th>
<th>Count</th>
<th>Cond.</th>
<th>Unit Mask</th>
<th>Prog Ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>= 0</td>
<td>MMA</td>
<td>0xABCD</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2020 MYTHIC. ALL RIGHTS RESERVED.
Graph Compiler Handles Flow Scoreboard Configuration

Open Neural Network Exchange (ONNX)

MatMul

Previous Node
Constant Weight Matrix
Next Node

Each graph element has a Mythic Flow Scoreboard translation

Previous Operation
FIFO
(Graph Edge)

(A: FIFO)

New Data Token
(ID0 Decrement 1)

MMA Operation
(Graph Node)

(B: Weight Config)

Space Free Token
(ID1 Decrement 1)

FIFO
(Graph Edge)

Next Operation

© 2020 MYTHIC. ALL RIGHTS RESERVED.
Graph Compiler Optimizes The Graph Throughput

Network-on-Chip is ideal for this style architecture
- Traffic is often to an adjacent tile
- Parallelism increases only affect local resources

Bottleneck Identified

Parallelize Across More Tiles

ResNet-18 on a 30 Tile Mythic IPU

Single-Step Optimized ResNet-18
Mythic SDK Enables Developers With The Latest Frameworks

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

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

© 2020 MYTHIC. ALL RIGHTS RESERVED.
Towards using OpenMP to program Mythic IPUs

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
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][i])
            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])
                dgemm(a[k][i], a[k][j], a[j][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

Alexnet taken from Khronos NNEF specification
#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];    // Transpose 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.
}
/* 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; oc<112; oc++)
                for (int ic=0; ic<112; ic++)
                    for (int f0=0; f0<3; f0++)
                        for (int f1=0; f1<3; f1++)
                            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; oc<112; oc++)
                for (int ic=0; ic<112; ic++)
                    for (int f0=0; f0<3; f0++)
                        for (int f1=0; f1<3; f1++)
                            dst[k0][k1][:112] += src[k0][k1][f0][f1][:112] * W[f0][f1][:112][:112];
    // MYTHIC_MMADOT()
}

© 2020 MYTHIC. ALL RIGHTS RESERVED.
Leverage FSB for Data Flow across Tasks assigned to tiles

```c
void conv2d_mythic_mmadot()
{
    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

```c
#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 IPUs.
OpenMP Task/Target with depend

```c
#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
```

GStreamer standard plugins
- rtp
-capsfilter
-queue
-htph264depay
-h264parse
-avdec_h264
-video-rate
-videoconvert

Mythic-specific plugins
- SSD-preprocess
- mythic-plugin
- SSD-postprocess
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.