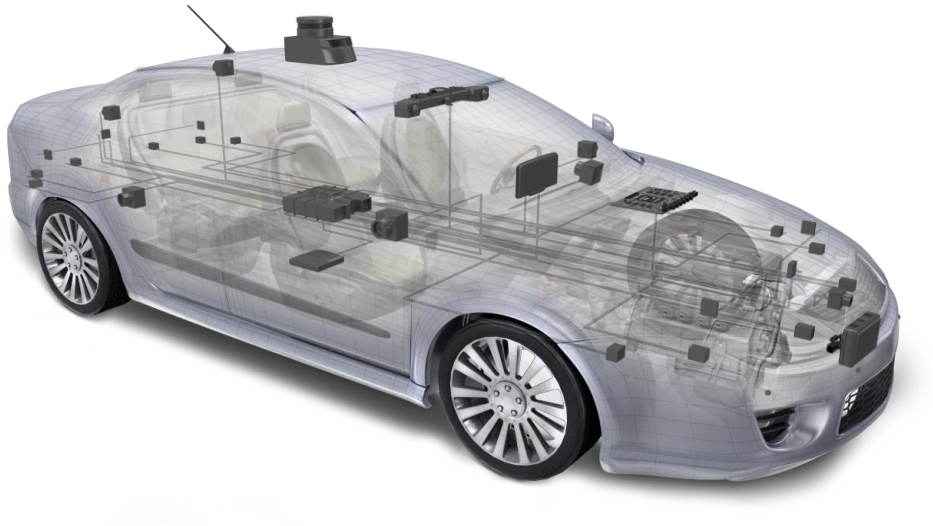
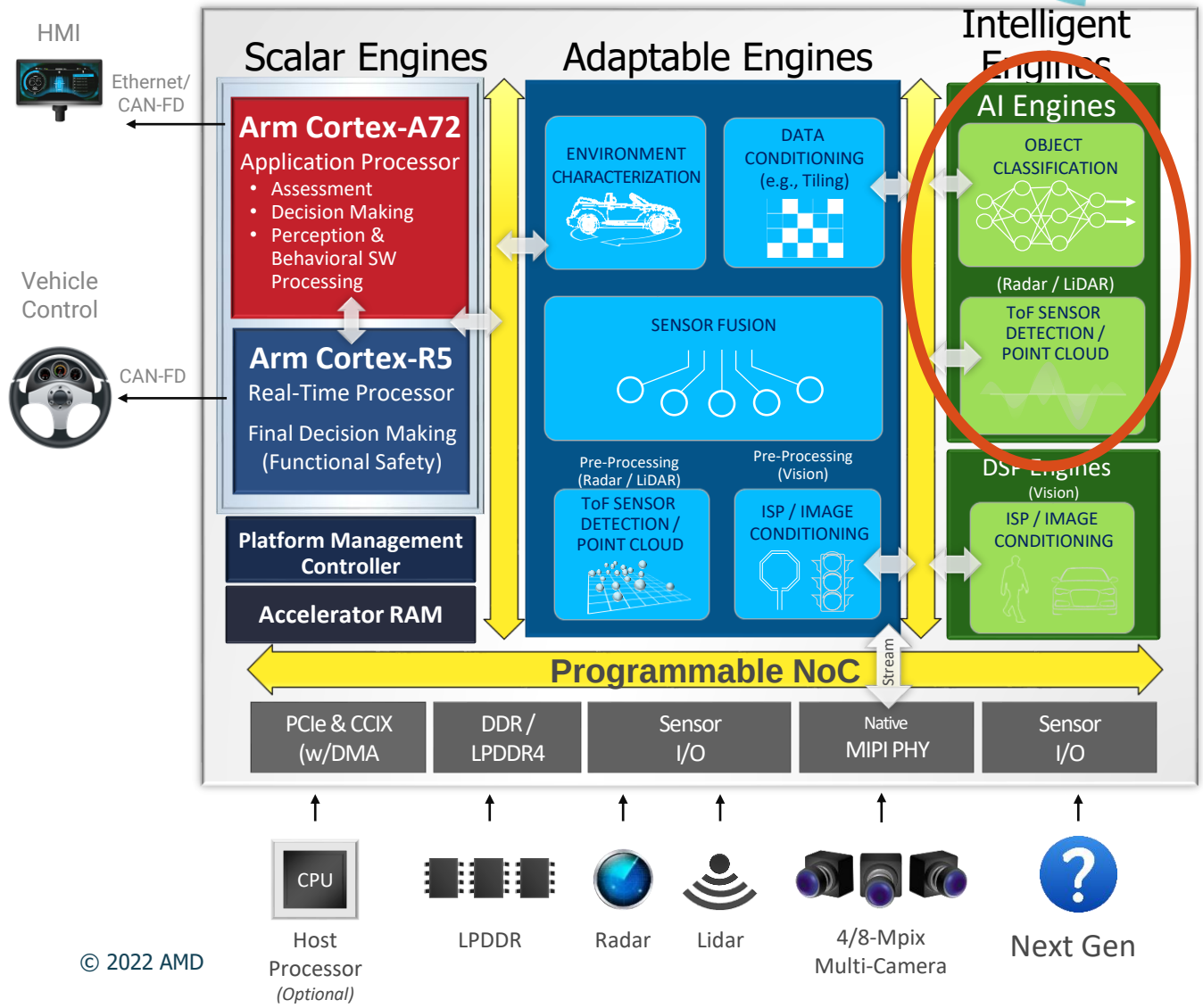
Programming Vision Pipelines on AMD's AI Engines

Kristof Denolf (Principal Engineer)
Bader Md Alam (Director SW Engineering)
AMD

Versal is a Heterogenous Chip Well Suited for Vision





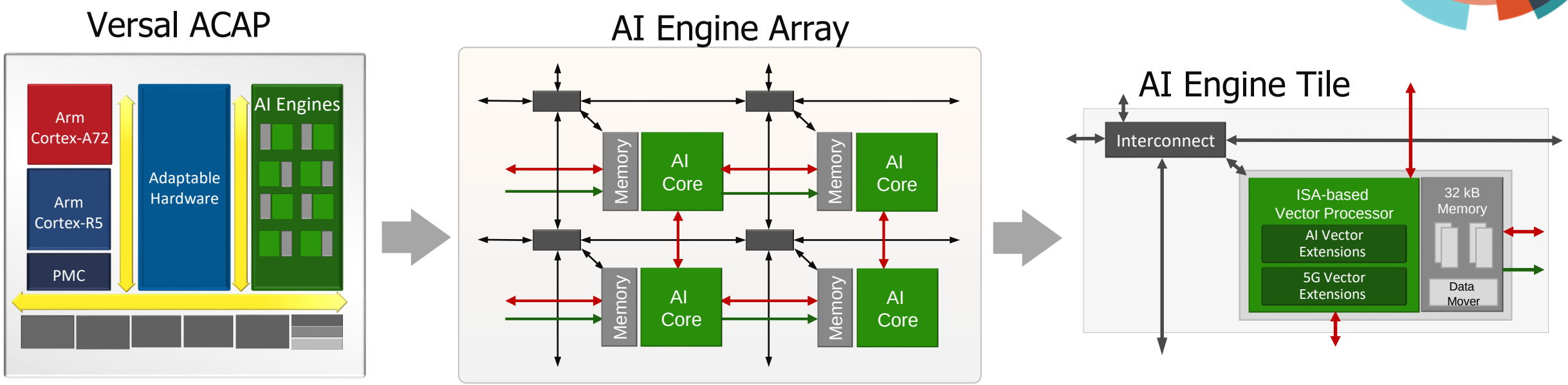
Let's focus on the AI Engines

Agenda

- AI Engine Technology Introduction
  - Compute Capabilities of the AI Engine
  - Data Movement
- Vitis Vision:
  - Library Overview
  - Programming Vision Pipelines with Vitis
- Sneak Preview AIE-ML
- Conclusion

AI Engine Technology Introduction

Versal and AI Engine Terminology



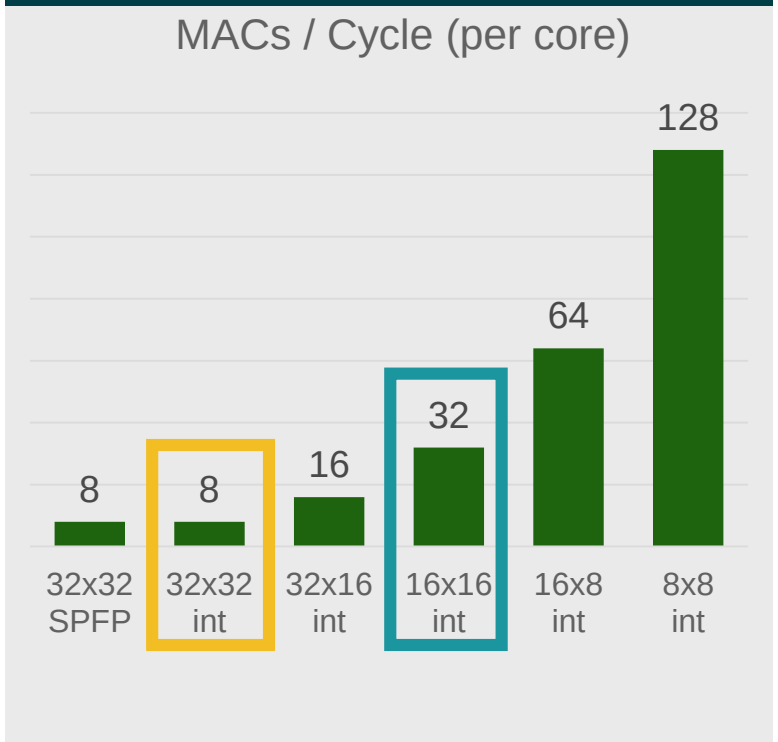
Data type	GMACs per AI Engine	x 128 (MACs/s)	X400 (MACs/s)
int8	128	16 T	51 T
int16	32	4 T	13 T
{foat,int}32	8	1 T	3 T

128-400 1GHz AI Engines (Versal Core)

- AI Engine is a VLIW vector processor
- 32 kB memory, locks and data movers
- Directly connected to its neighbors
- Fully connected through AXI Stream interconnect
- MAC = 2 Ops

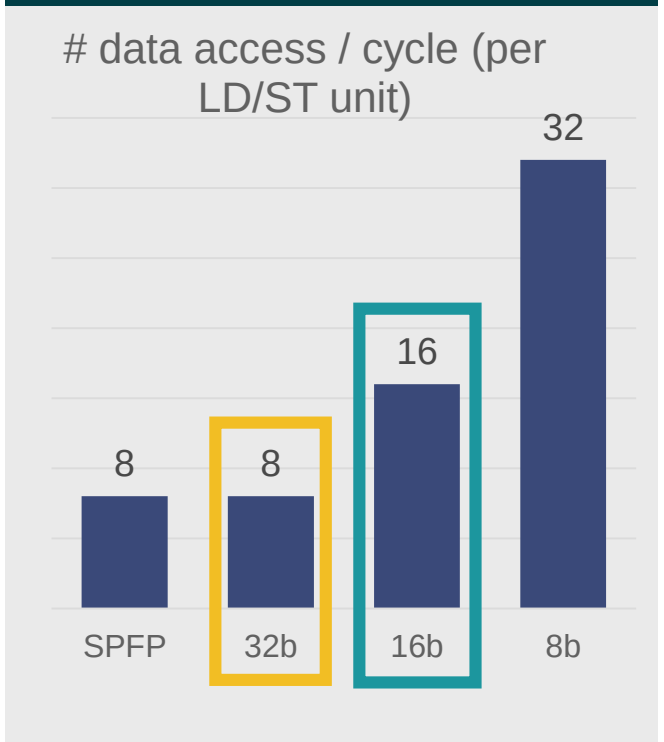
Multi-Precision Support Enables Different Pixel Depths

AI Data Types



Measured results
Vectorization Example

Local Memory Access



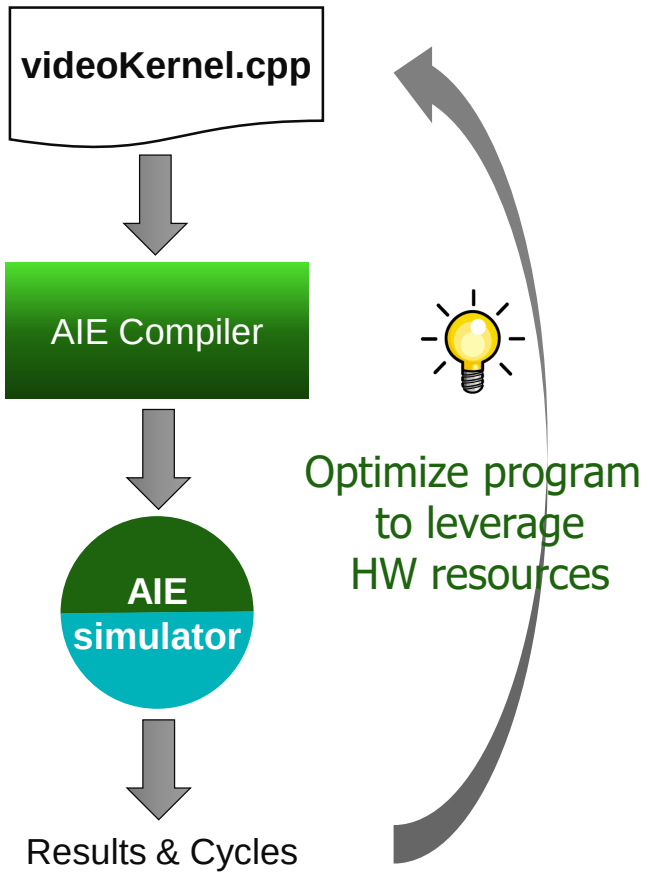
Each AI Engine has:
- 2 x 256b LD units
- 1 x 256b ST

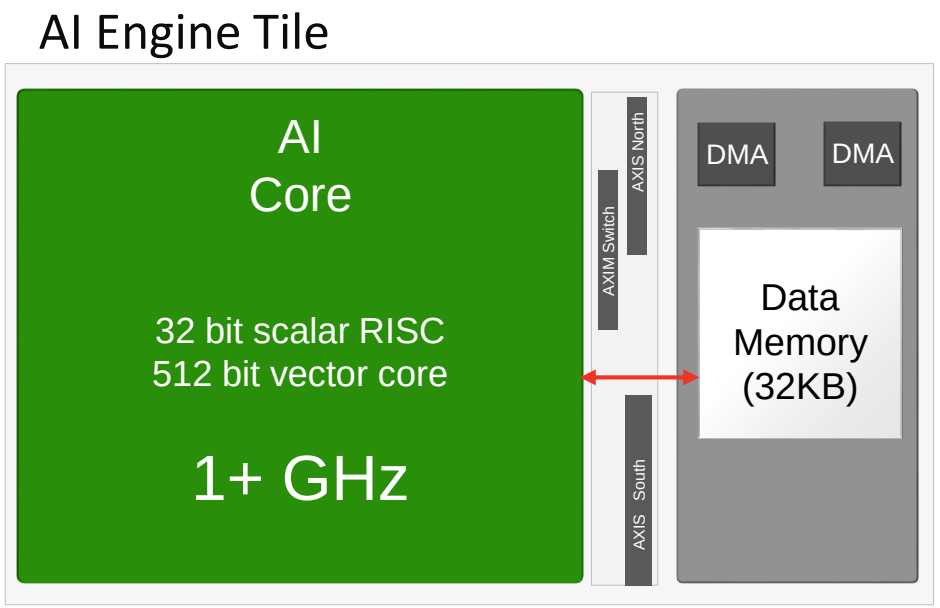
Data reuse needed to match memory bandwidth with 100% MAC utilization

Config	Data reuse	Coeff Reuse
32x32	1x	1x
32x16	2x	1x
16x16	2x	2x
16x8	4x	2x
8x8	4x	4x

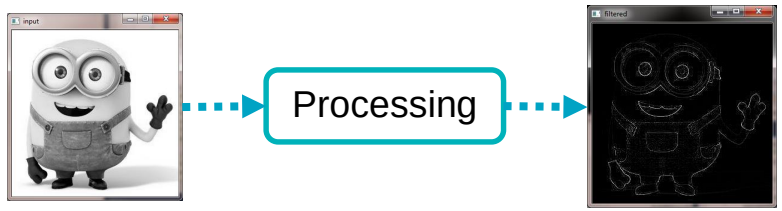
- More compute with smaller datatypes
- Data reuse to enable maximum vector compute

AI Engine: SW Programmable Signal Processor



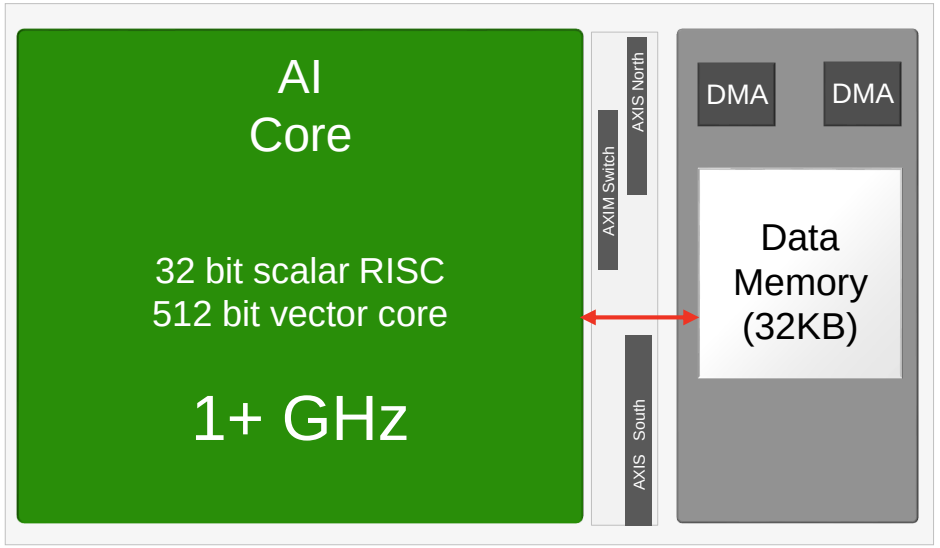


AI Engine: SW Programmable Signal Processor with Zero Loop Overhead on Counters and Buffer Auto Increment



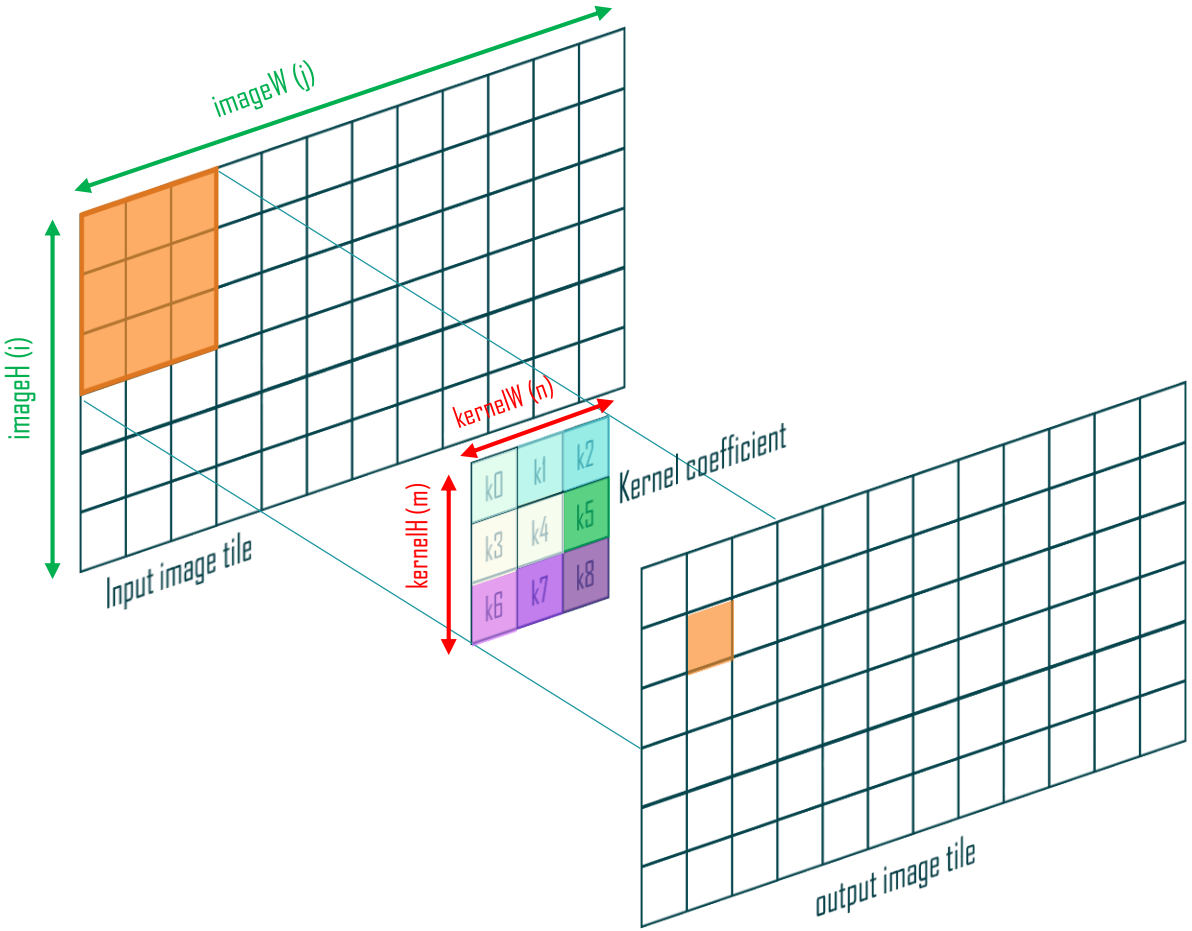

```
int32 *inDataMemory;
int32 *outDataMemory;
aie::vector<int32,16> vectorOfData;

loop(expression) {
  loop(expression) {
    vectorOfData = *inDataMemory++;
    processing on vectorOfData;
    *outDataMemory++ = vectorOfResults;
    }
  }
```

AI Engine Tile



Filter2D – Basic Algorithm

32b data x 32b coefficients



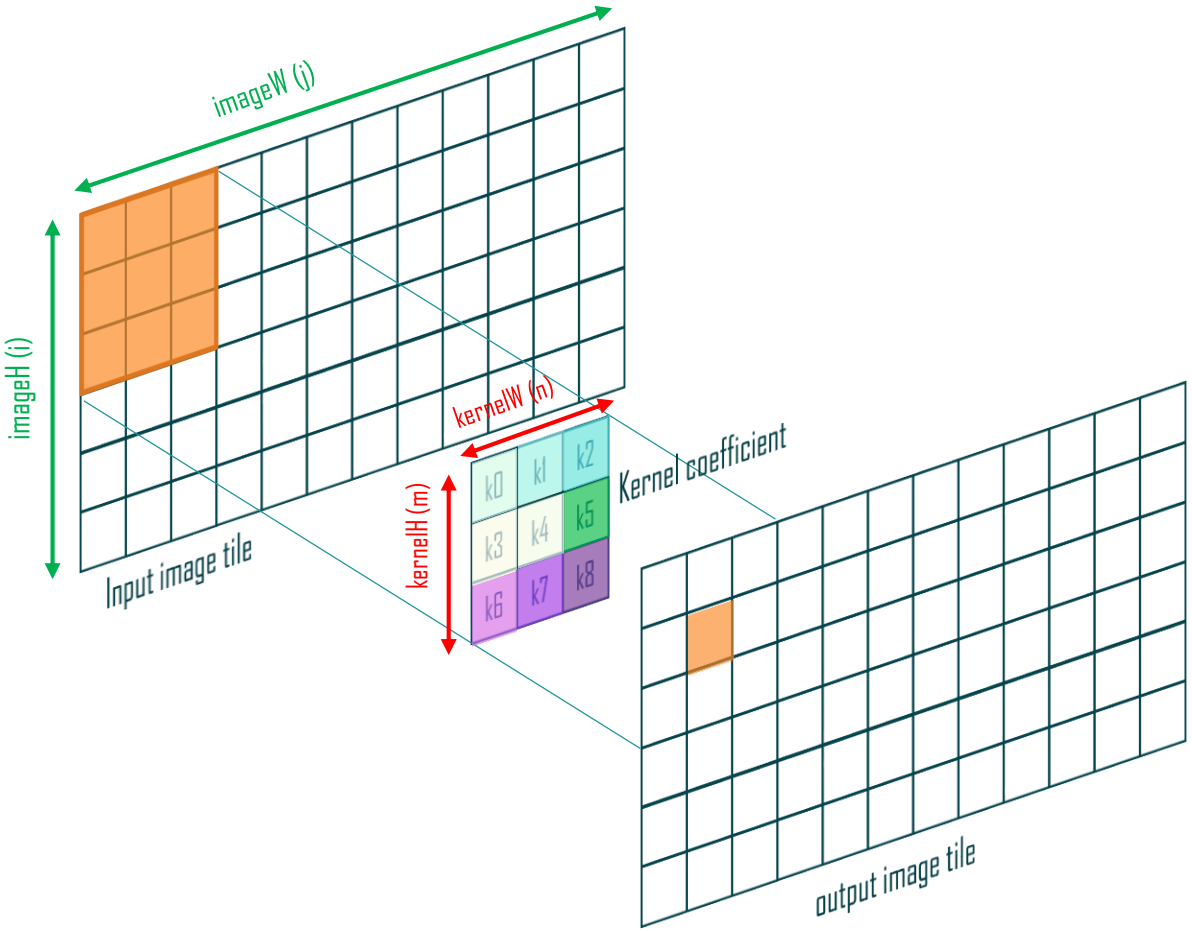

```
int32 *img_in;

for(int i=0; i<imageH; i++) {
  for(int j=0; j<imageW; j++) {
    int32_t accum = 0;
    for(int m = 0; m < kernelH; m++){
      for(int n = 0; n < kernelW; n++) {
        accum += kernel_coeff[m*kernelW+n]*
                 img_in[(m+i)*imageW + (j+n)];
      }
    }
    img_out[i*image_width + j ] = accum; } }
```

Complexity

- O(N,k^2)
- N = Image Size, k = Kernel Size

Filter2D – Unroll Inner Loops (Prepare for Vectorization)

32b data x 32b coefficients

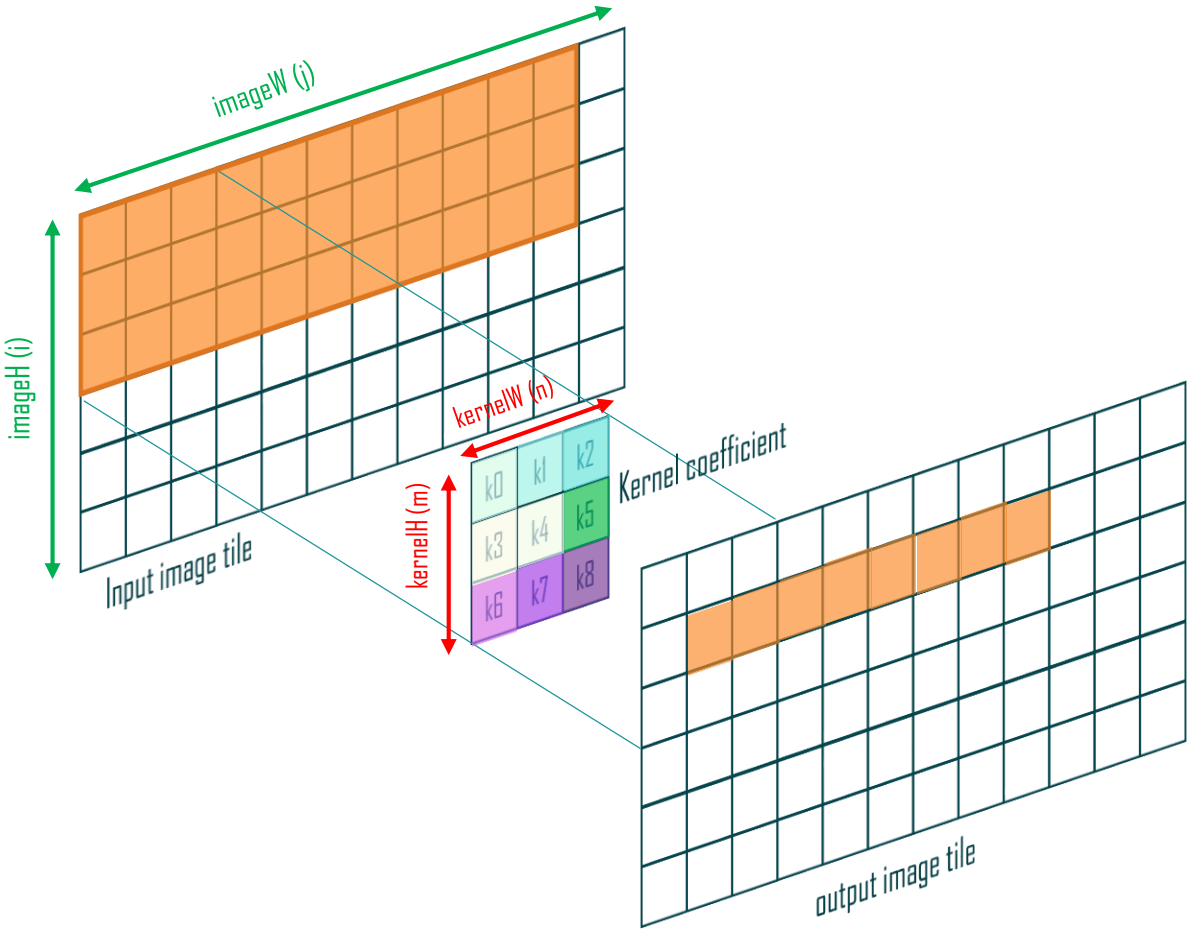



```
int32 *img_in;

for(int i=0; i<imageH; i++) {
  for(int j=0; j<imageW; j++) {
    int32_t accum = 0;
    accum  = kernel_coeff[0]*img_in[(0+i)*imageW+(j+0)];
    accum += kernel_coeff[1]*img_in[(0+i)*imageW+(j+1)];
    accum += kernel_coeff[2]*img_in[(0+i)*imageW+(j+2)];
    accum += kernel_coeff[3]*img_in[(1+i)*imageW+(j+0)];
    accum += kernel_coeff[4]*img_in[(1+i)*imageW+(j+1)];
    accum += kernel_coeff[5]*img_in[(1+i)*imageW+(j+2)];
    accum += kernel_coeff[6]*img_in[(2+i)*imageW+(j+0)];
    accum += kernel_coeff[7]*img_in[(2+i)*imageW+(j+1)];
    accum += kernel_coeff[8]*img_in[(2+i)*imageW+(j+2)];

    img_out[i*image_width + j ] = accum; }
}
```

Unrolled (for 3x3 kernel)

Filter2D – Vectorize by 8 in Horizontal Dimension

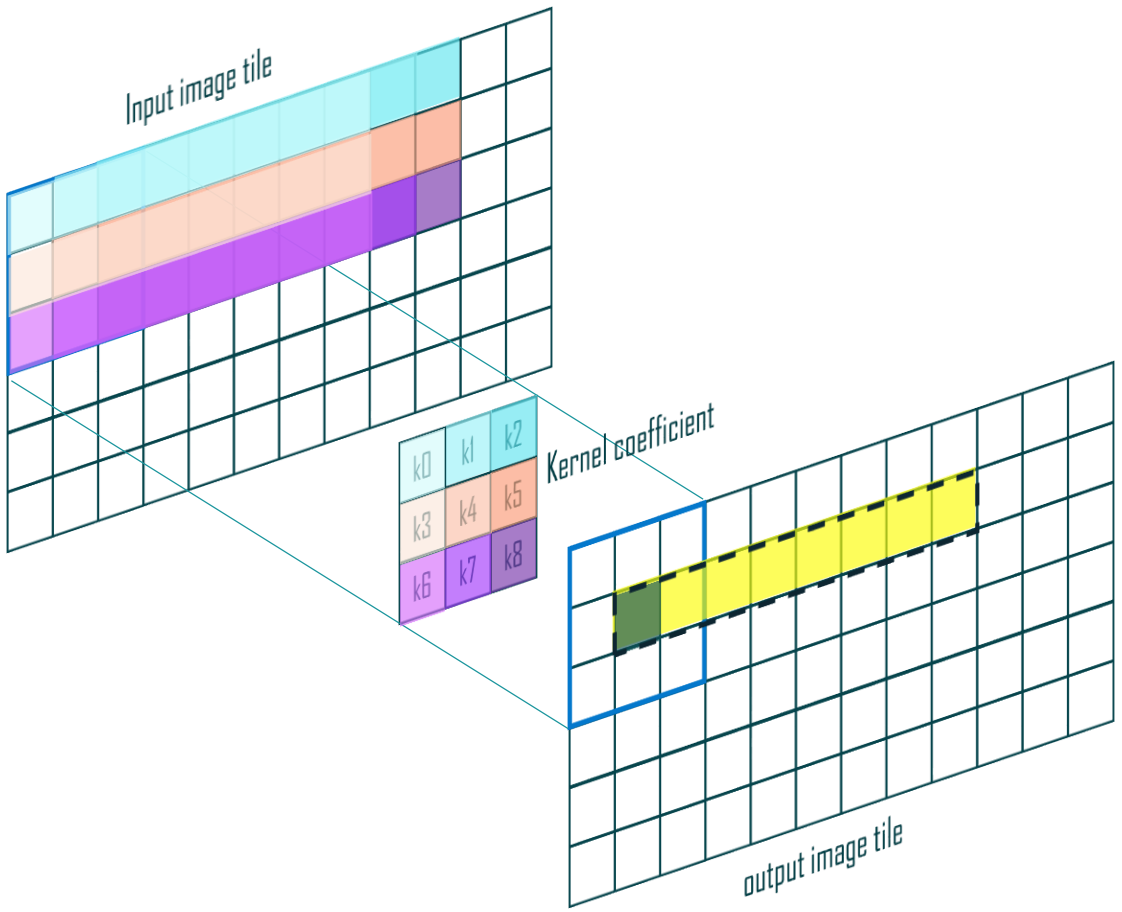
32b data x 32b coefficients



- Scalar Reference Solution (32b data and 32b coefficients)

```
int32 *img_in;

for(int i=0; i<imageH; i++) {
  for(int j=0; j<imageW; j+=8)  {
    vector<int32_t,8> accum8 = 0;
    accum8  = kernel_coeff[0]*img_in[r1:0..7];
    accum8 += kernel_coeff[1]*img_in[r1:1..8];
    accum8 += kernel_coeff[2]*img_in[r1:2..9];
    accum8 += kernel_coeff[3]*img_in[r2:0..7];
    accum8 += kernel_coeff[4]*img_in[r2:1..8];
    accum8 += kernel_coeff[5]*img_in[r2:2..9];
    accum8 += kernel_coeff[6]*img_in[r2:0..7];
    accum8 += kernel_coeff[7]*img_in[r2:1..8];
    accum8 += kernel_coeff[8]*img_in[r2:2..9];

    img_out[i*image_width + j ] = accum8; }
}
```

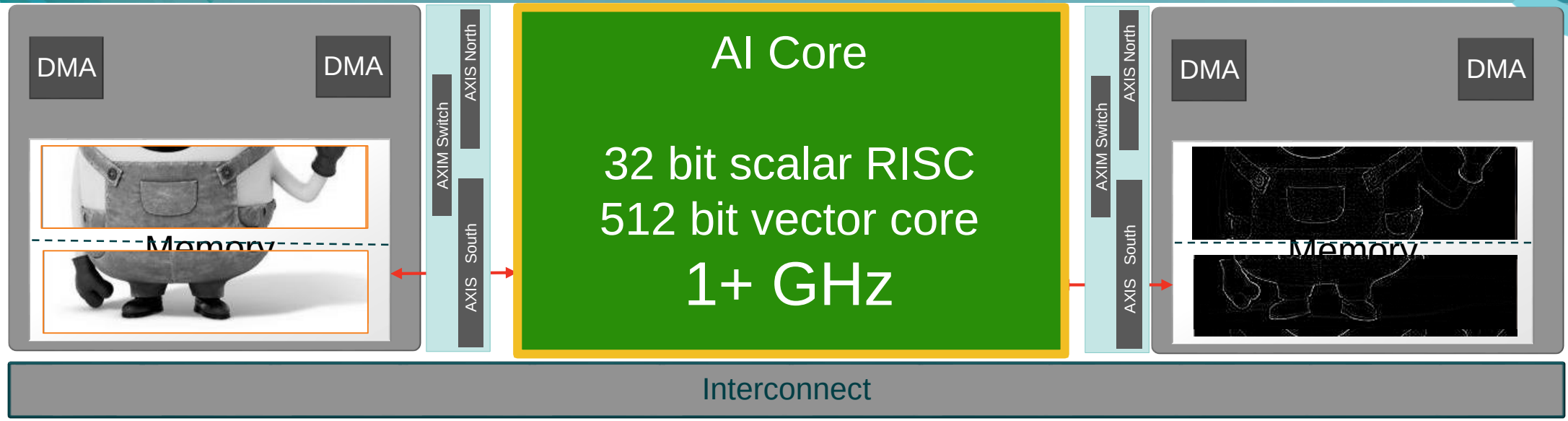
Vectoring with Factor 8 while Exploiting Vector Register Data Reuse through Select

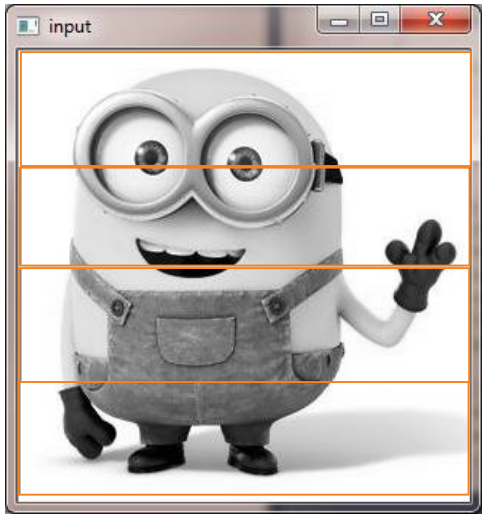


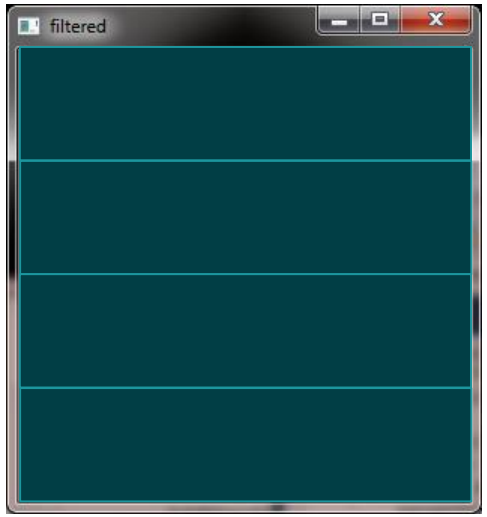
New Inner loop pseudo code

```
acc  = mul(coeff, c_sel1, data_buf, d_sel1);
acc += mul(coeff, c_sel2, data_buf, d_sel2);
acc += mul(coeff, c_sel3, data_buf, d_sel3);
acc += mul(coeff, c_sel4, data_buf, d_sel4);
acc += mul(coeff, c_sel5, data_buf, d_sel5);
acc += mul(coeff, c_sel6, data_buf, d_sel6);
acc += mul(coeff, c_sel7, data_buf, d_sel7);
acc += mul(coeff, c_sel8, data_buf, d_sel8);
acc += mul(coeff, c_sel9, data_buf, d_sel9);
```

AI Engine (Array) is Built for Parallel Data Movement and Compute

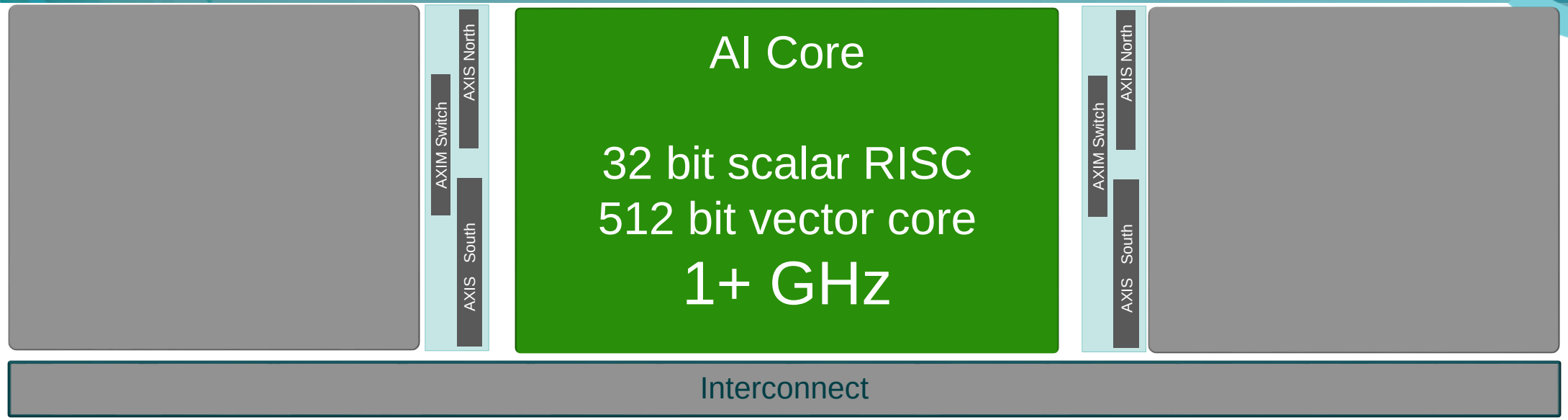




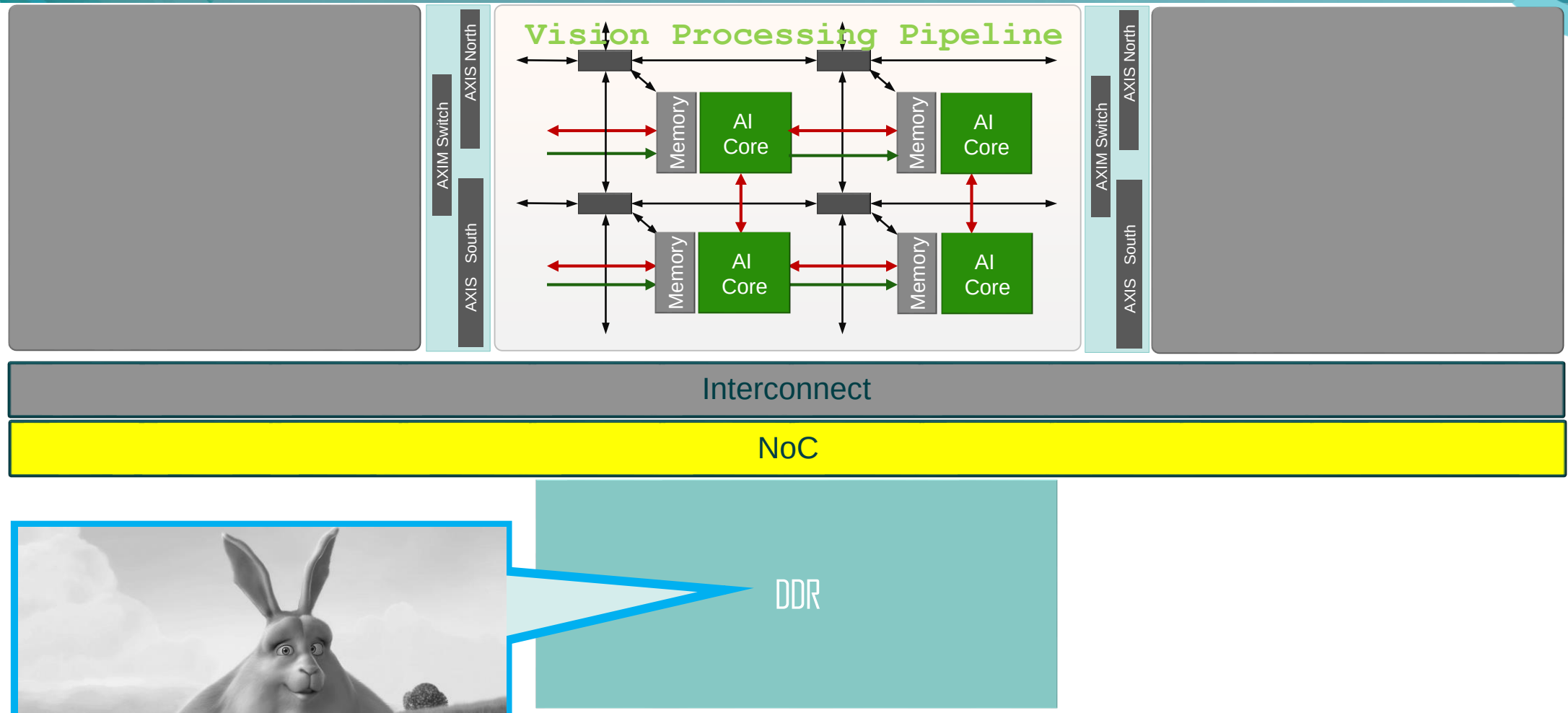


- Data push system
- Control flow support → data flow style implementations

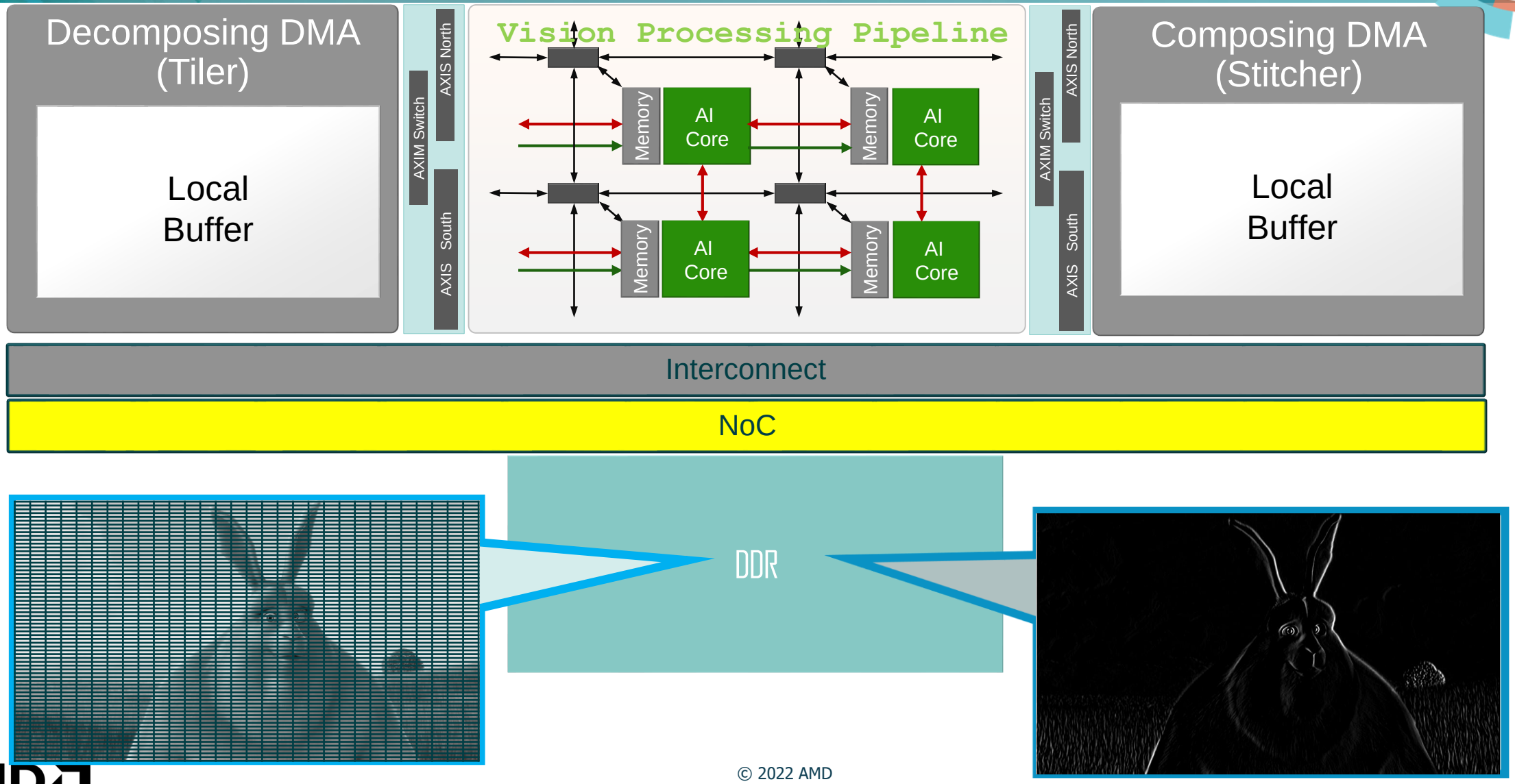
Zoom out to System Level



Zoom out to System Level



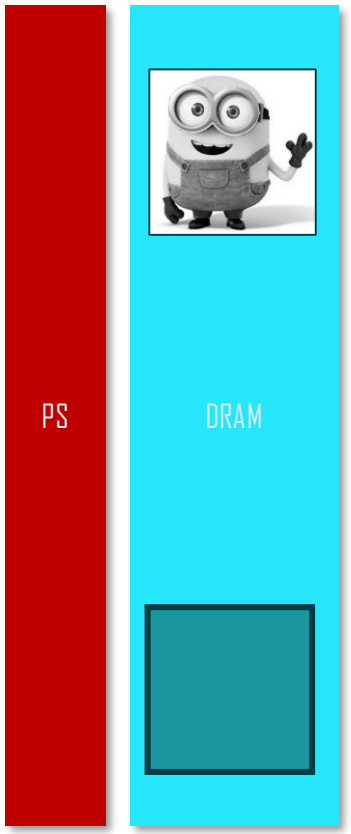
Vision Processing Graph Exploits Specialized Data Movement

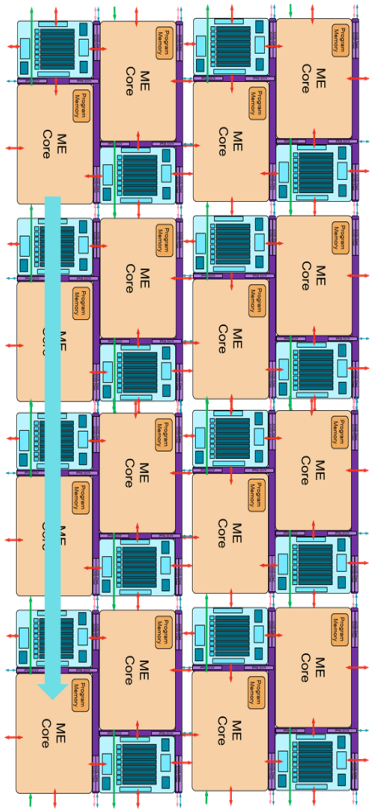


Vitis Vision: Library Overview, Programming a Vision Pipeline and Tools

What is in the AI Engine Vision Library?

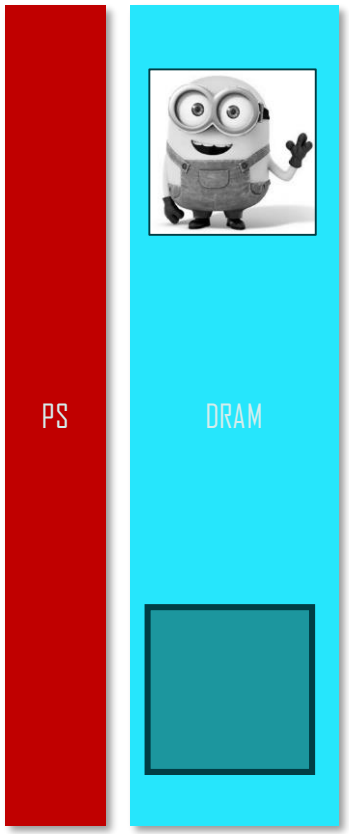
```
cv2.filter2D(img,-1,kernel,dst)
```

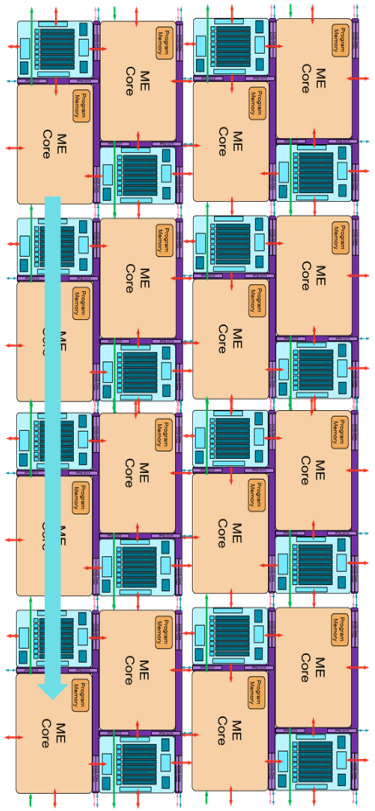




What is in the AI Engine Vision Library?

```
cv2.filter2D(img,-1,kernel,dst)
```



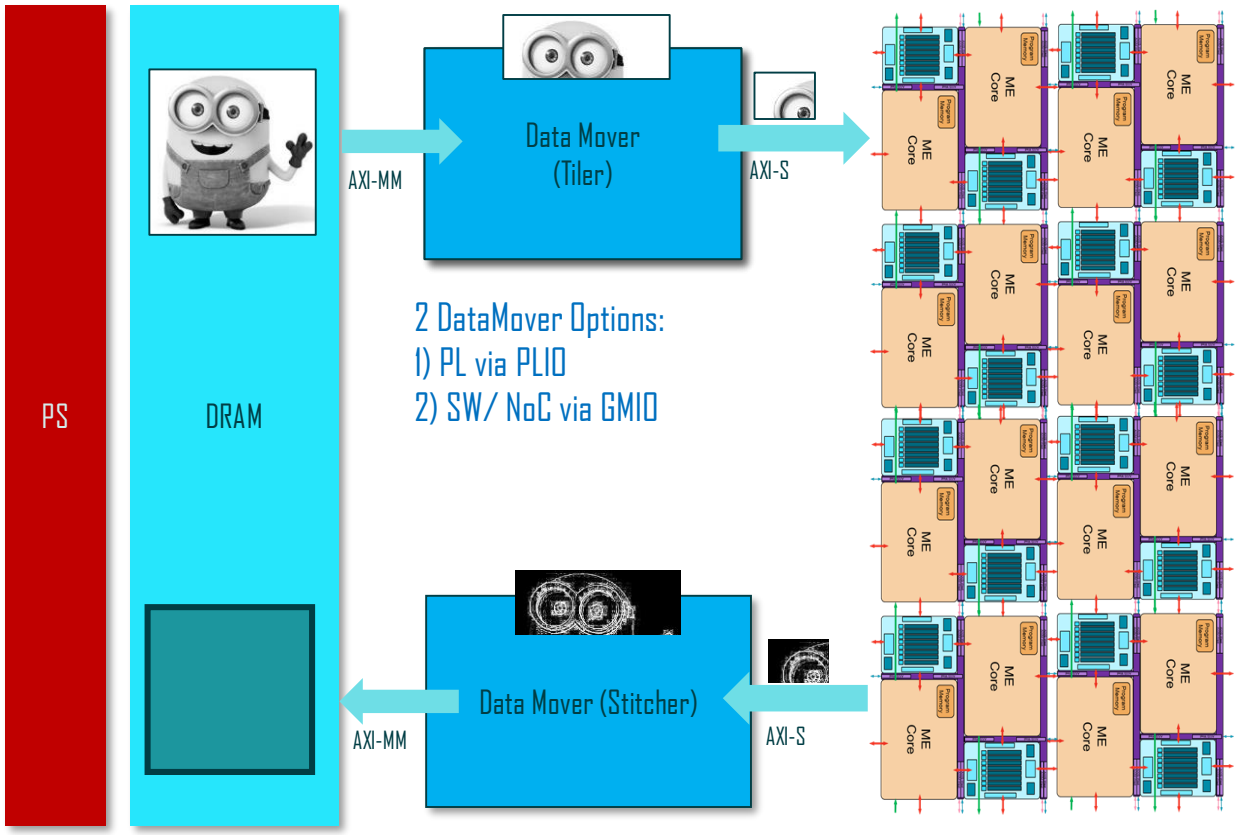


Host code

What is in the AI Engine Vision Library?

data-movement

```
cv2.filter2D(img,-1,kernel,dst)
```



Ease-of-Use – High level abstraction for data movement

Code to define DataMover

```
xF::xfcvDataMover<xF::TILER, ...> tiler(1,1);
xF::xfcvDataMover<xF::STITCHER, ...> stitcher;
```

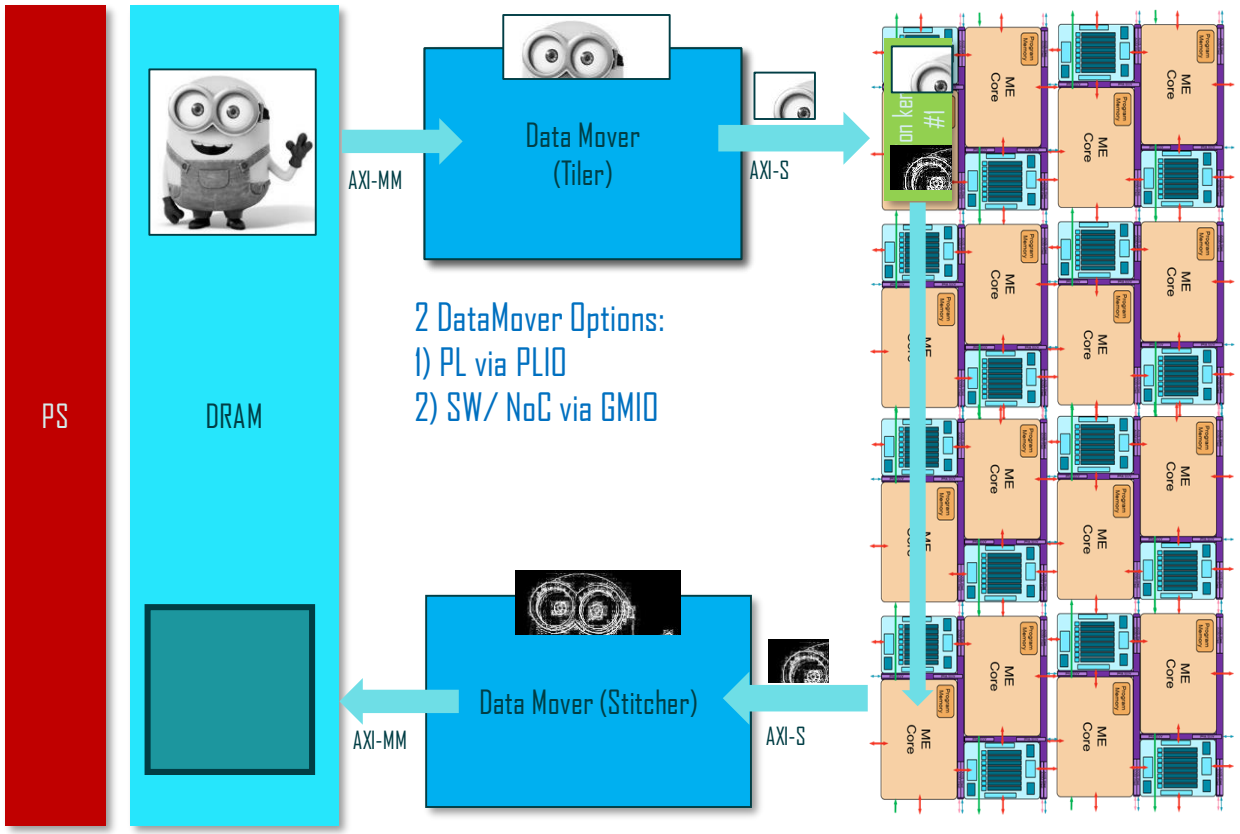
Host code

What is in the AI Engine Vision Library?

data-movement

AI Engine vision kernels

```
cv2.filter2D(img,-1,kernel,dst)
```



Ease-of-Use – High level abstraction for data movement

Code to define DataMover

```
xF::xfcvDataMover<xF::TILER, ...> tiler(1,1);
xF::xfcvDataMover<xF::STITCHER, ...> stitcher;
```

Graph code for kernel

```
myGraph filter_graph();
```

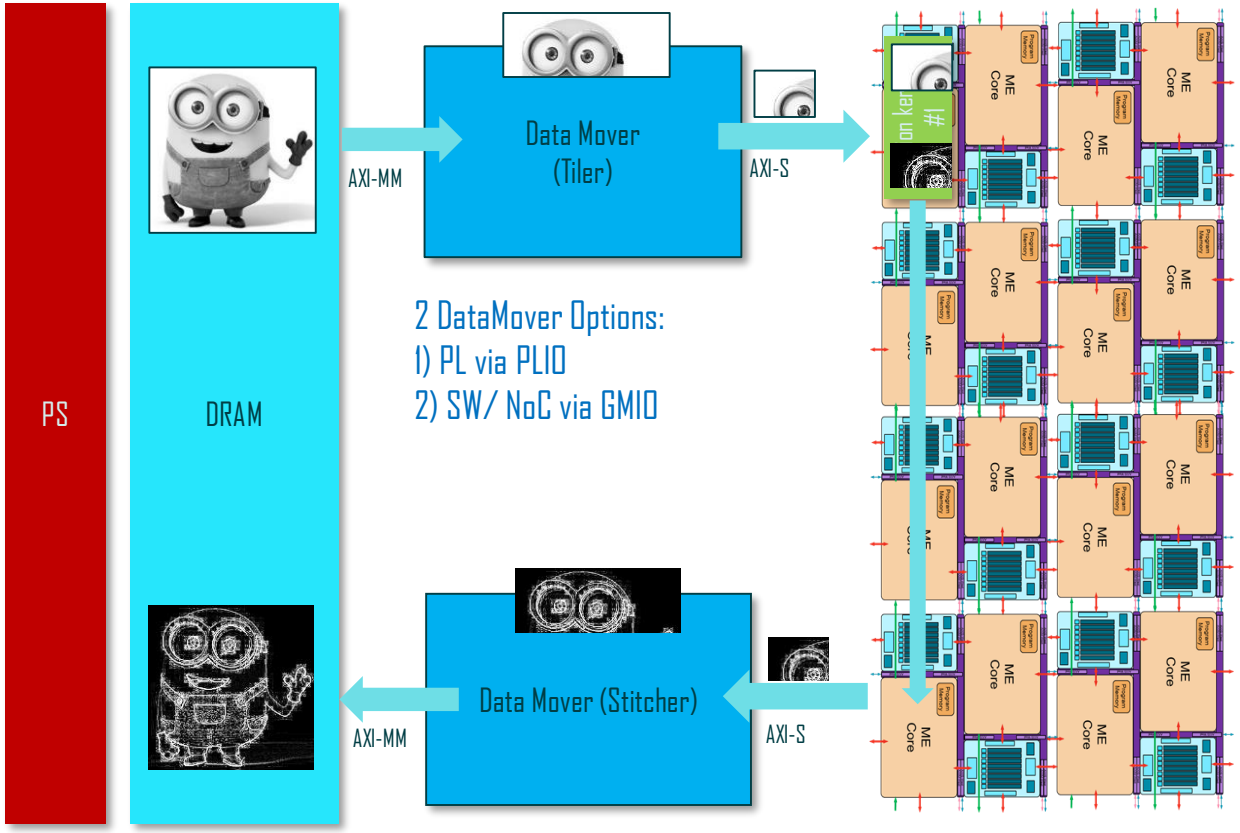
Host code

What is in the AI Engine Vision Library?

data-movement

AI Engine vision kernels

```
cv2.filter2D(img,-1,kernel,dst)
```



Ease-of-Use – High level abstraction for data movement

Code to define DataMover

```
xF::xfcvDataMover<xF::TILER, ...> tiler(1,1);
xF::xfcvDataMover<xF::STITCHER, ...> stitcher;
```

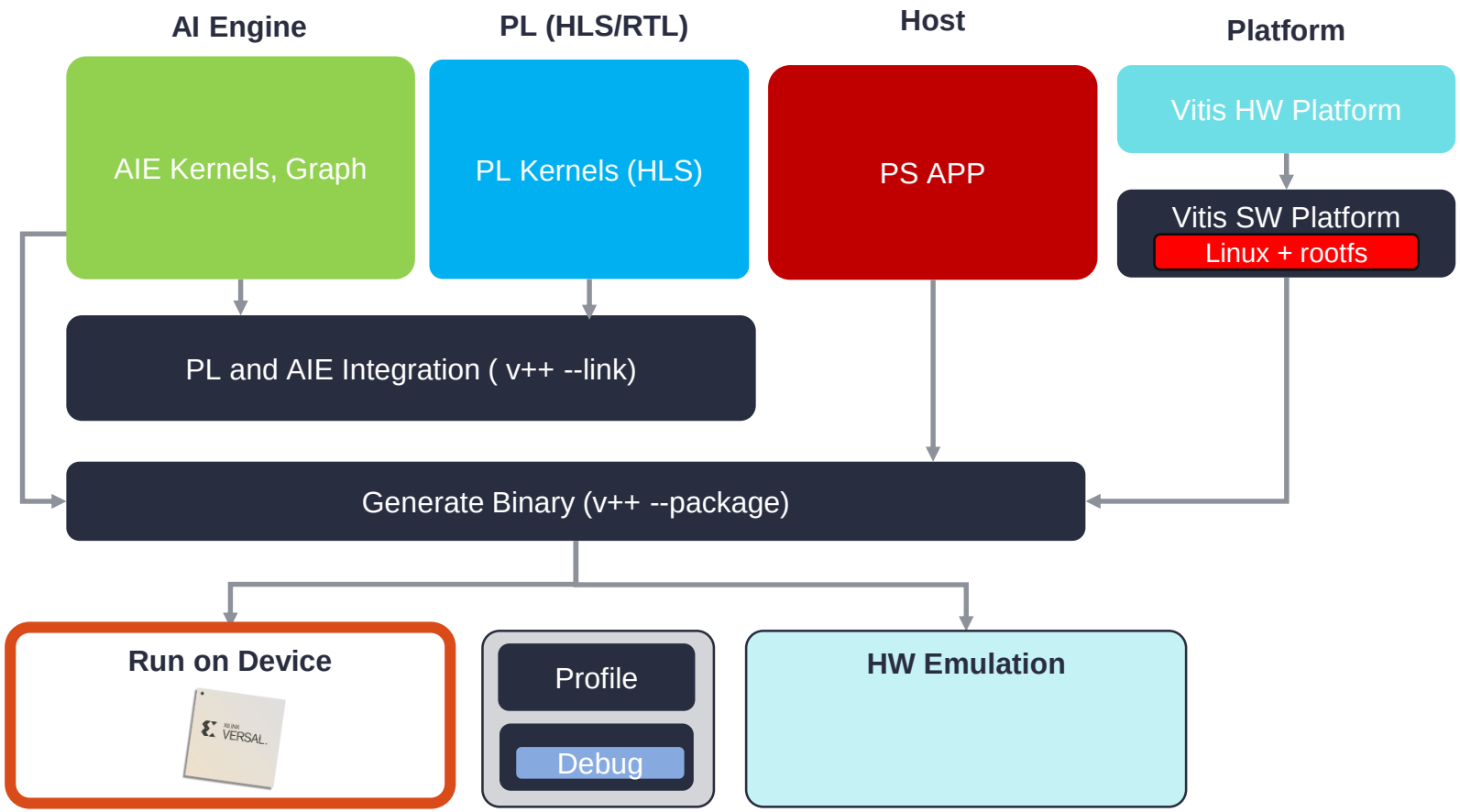
Graph code for kernel

```
myGraph filter_graph();
```

Host code to call datamover & run graph

```
auto tiles_sz = tiler.host2aie_nb(src_hndl, srcImageR.size());
stitcher.aie2host_nb(dst_hndl, dst.size(), tiles_sz);
std::cout << "Graph run(1)\n");
filter_graph[j].run(1);
filter_graph[j].wait();
tiler.wait();
stitcher.wait();
```

Vitis Tool Overview



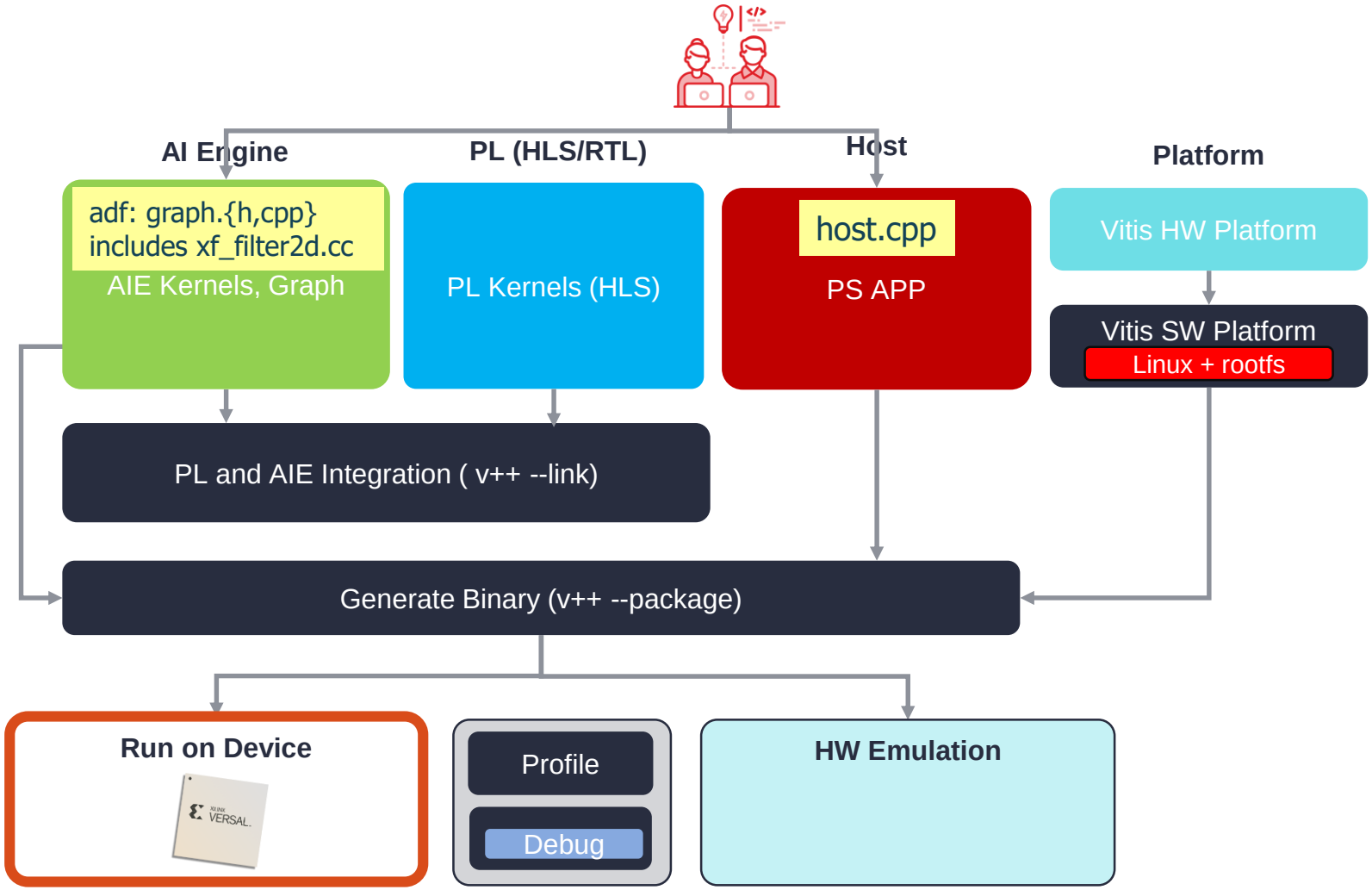
data-movement

AI Engine vision kernels

Host code

Vitis Tool Overview Slide

```
cv2.filter2D(img,-1,kernel,dst)
```

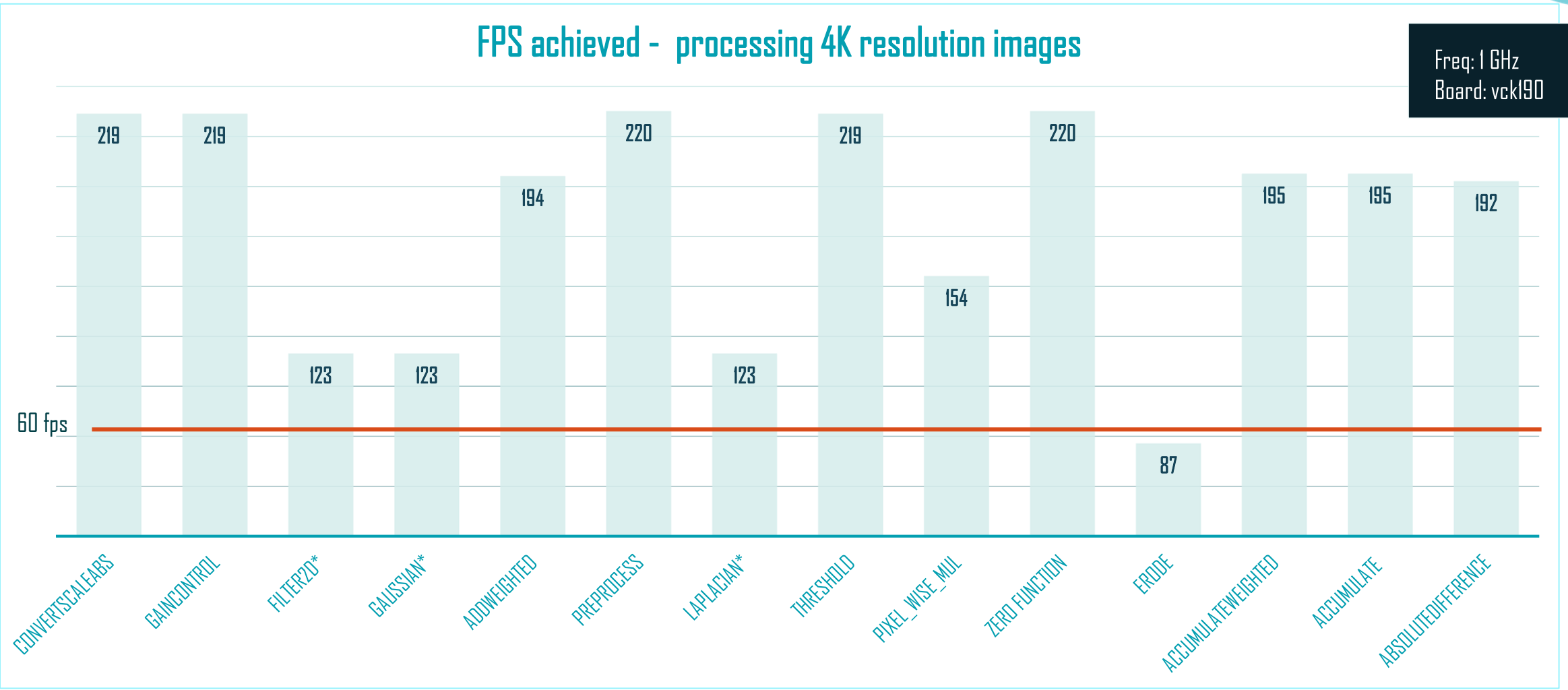


data-movement

AI Engine vision kernels

Host code

Library of Optimized Vision Kernels – 1x AI Engine Core Performance



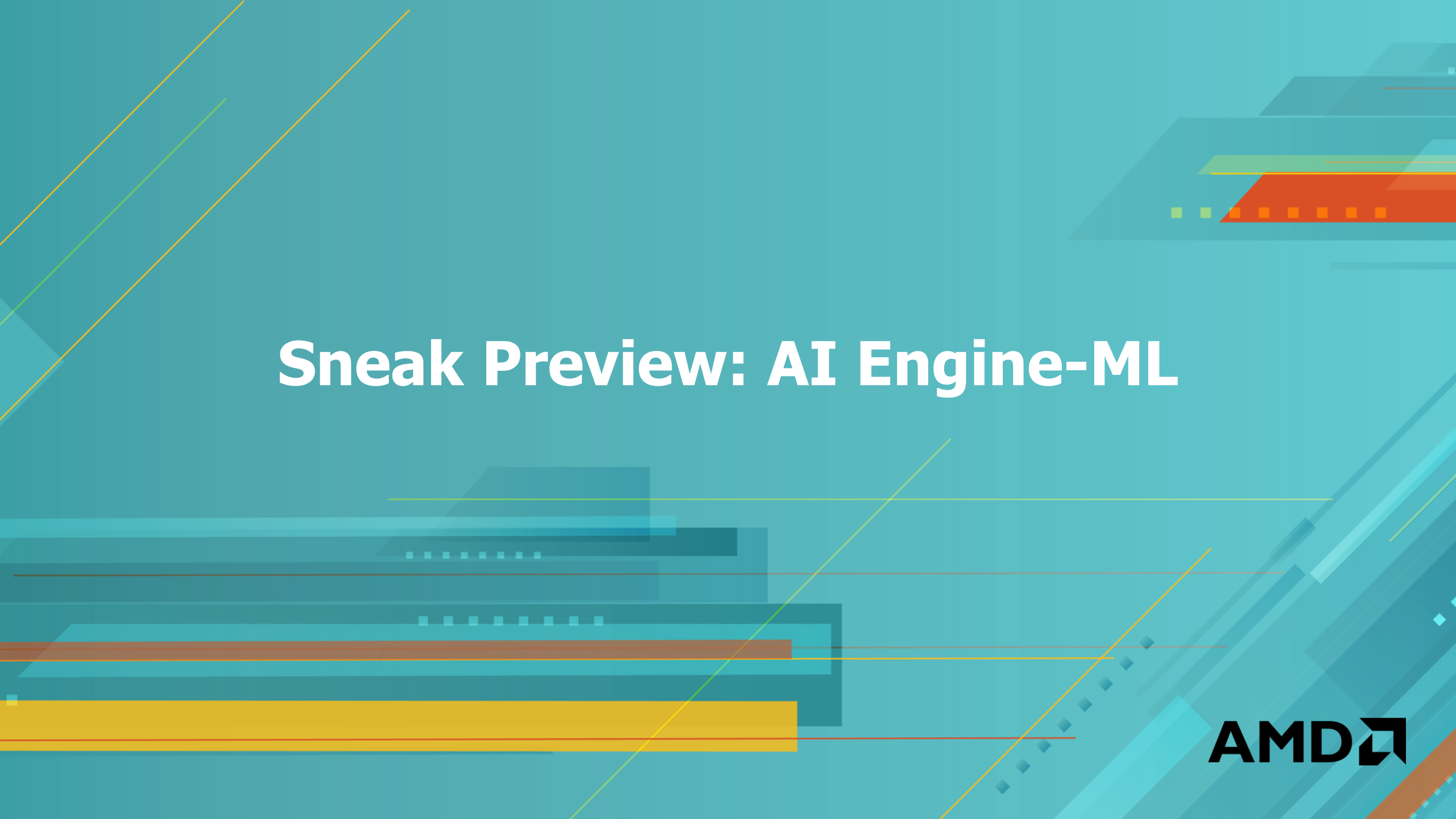
Vitis Vision Library: AI Engine Portfolio

Vitis Vision Lib:
AI Engine Portfolio

2021.1 / planned

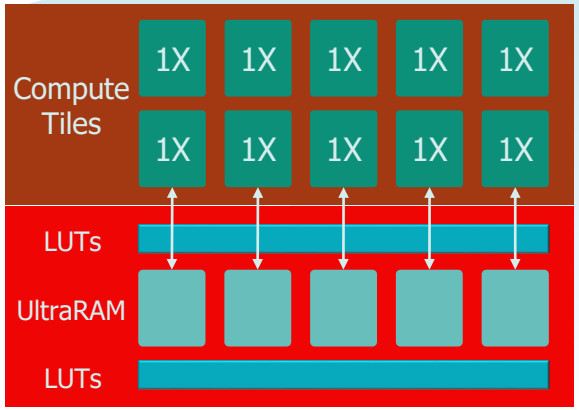
Basic Functionality	DNN (X of X+ML)	Image Sensor Processing (ISP)	Filters/ Others
		2D/3D Noise Reduction	
		Mono, RGB-IR Debayering	
	Bicubic Resize	Tone Mappers	
	Background Matting	HDRFusion	Feature Extractors
	Mask Generation	Histogram Equalization	Remap
	IntersectionOfUnion	Quantization and Dithering	Warp
	Box-Sort	AWB	Stereo GBM
	NMS	AEC	Stereo LBM
	Crop/Patch	Gamma Correction	OTSU Thresholding
Absolute Difference	Channel Interleaving	BlackLevelCorrection	SeparableFilters
Accumulate Weighted		LenseShadingCorrection	
Accumulate	Normalization		filter2D
ConvertScaleAbs	Resize (Bilinear)	Gain Control	Gaussian Blur
PixelWiseMul	Thresholding	Defective Pixel Correction	Erode
ZeroFunction	ColorConversion	Debayering	Laplacian

Sneak Preview: AI Engine-ML



Intelligent Engines Optimized for Any Whole Vision AI Application

AI Engine Architecture



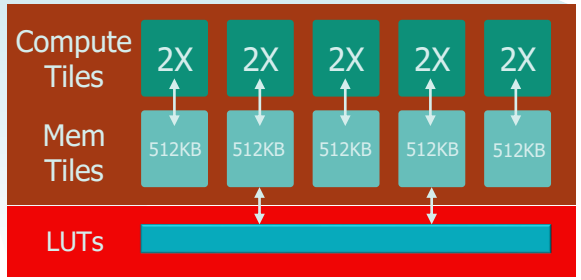
- Optimized for signal processing AND ML
- Flexibility for high performance DSP applications
- Native support for INT8, INT16, FP32

AIE	OPS / Tile	AIE-ML
256	INT4	1024
256	INT8	512
64	INT16	128
16	BFLOAT16	256
16	INT32	
16	FP32	42*

*Via software emulation

KB / Tile		
32	Data Memory	64
16	Program Memory	16

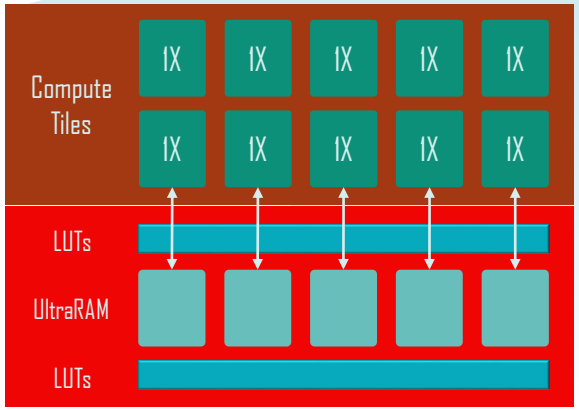
AIE-ML Architecture



- Optimized for ML Inference Applications
- Maximum AI/ML compute with reduced footprint
- Native support for INT4, INT8, INT16, bfloat16
- Fine grained sparsity HW optimization

Intelligent Engines Optimized for Any Whole Vision AI Application

AI Engine Architecture

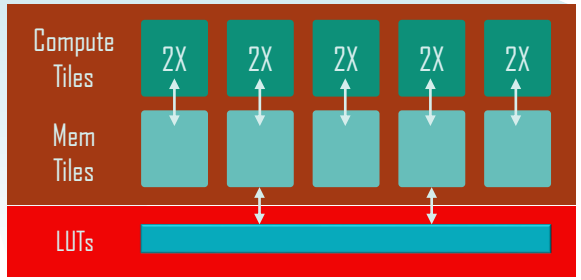


- Optimized for signal processing AND ML
- Flexibility for high performance DSP applications

KB / Tile		
16	INT32	
16	FP32	42*
32	Data Memory	64
16	Program Memory	16

*Via software emulation

AIE-ML Architecture



- Optimized for ML Inference Applications
- Maximum AI/ML compute with reduced footprint
- Native support for INT4, INT8, INT16, bfloat16

Versal AIE-ML offers 2X AI Performance per Watt

Conclusions

- The AI Engines of the Versal device support vison workloads by design
  - VLIW-vector processor with zero loop overhead and auto buffer increment
  - Vector compute in different bit depths covering essential vision operators
  - Concurrent data movement and compute in AI Engine array through DMAs
- Composing/Decomposing datamovers tiling into sub-images that fit local memory
- Supports streaming dataflow pipelines in the AI Engine array
- Growing kernel library covering both typical vision kernels and ISP kernels available in open source
- Vitis tools support easy programming of vision pipelines

Resources

Vitis (Vision) Resources

- **Vitis Vision**
  https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-vision.html
- **Github docs**
  https://xilinx.github.io/Vitis_Libraries/vision/2021.2/index.html
- **Github code**
  https://github.com/Xilinx/Vitis_Libraries/tree/master/vision
- **Vitis**
  https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

AI Engine Resources

- **AI Engines**
  https://www.xilinx.com/products/technology/ai-engine.html
- **Versal Core Product Family**
  https://www.xilinx.com/products/silicon-devices/acap/versal-ai-core.html
- **Versal AI Edge Product Family**
  https://www.xilinx.com/products/silicon-devices/acap/versal-ai-edge.html

2022 Embedded Vision Summit

Please visit our AMD boot

Thank You