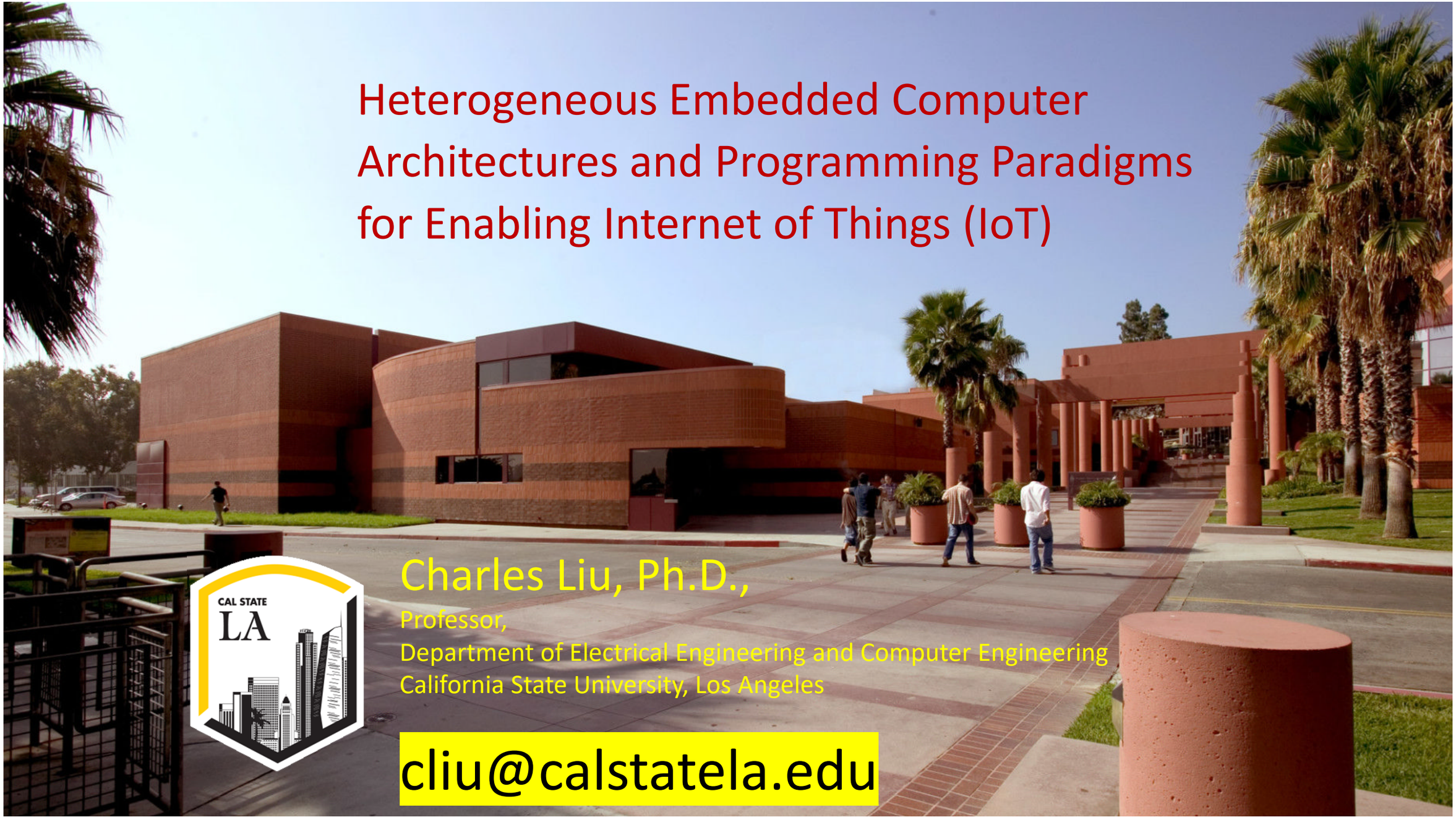


Heterogeneous Embedded Computer Architectures and Programming Paradigms for Enabling Internet of Things (IoT)

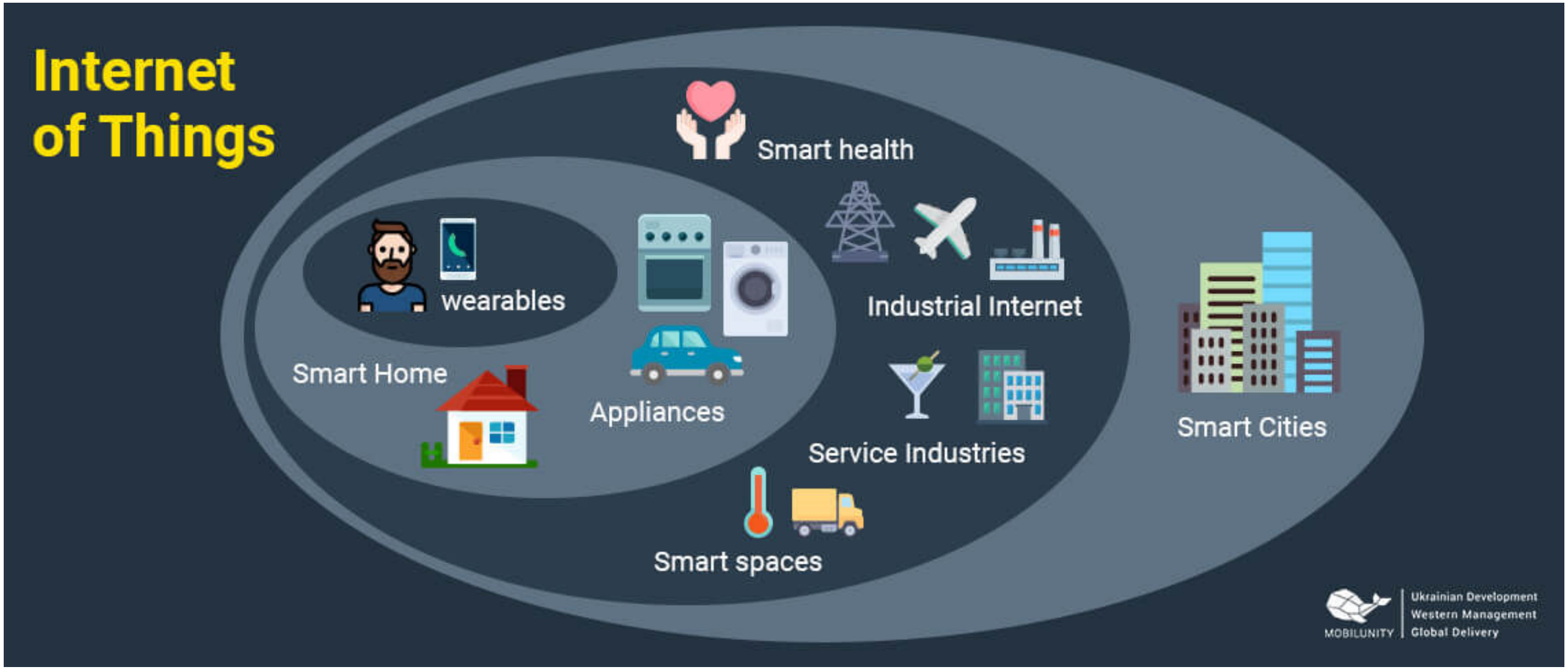
Charles Liu, Ph.D.,

Professor,

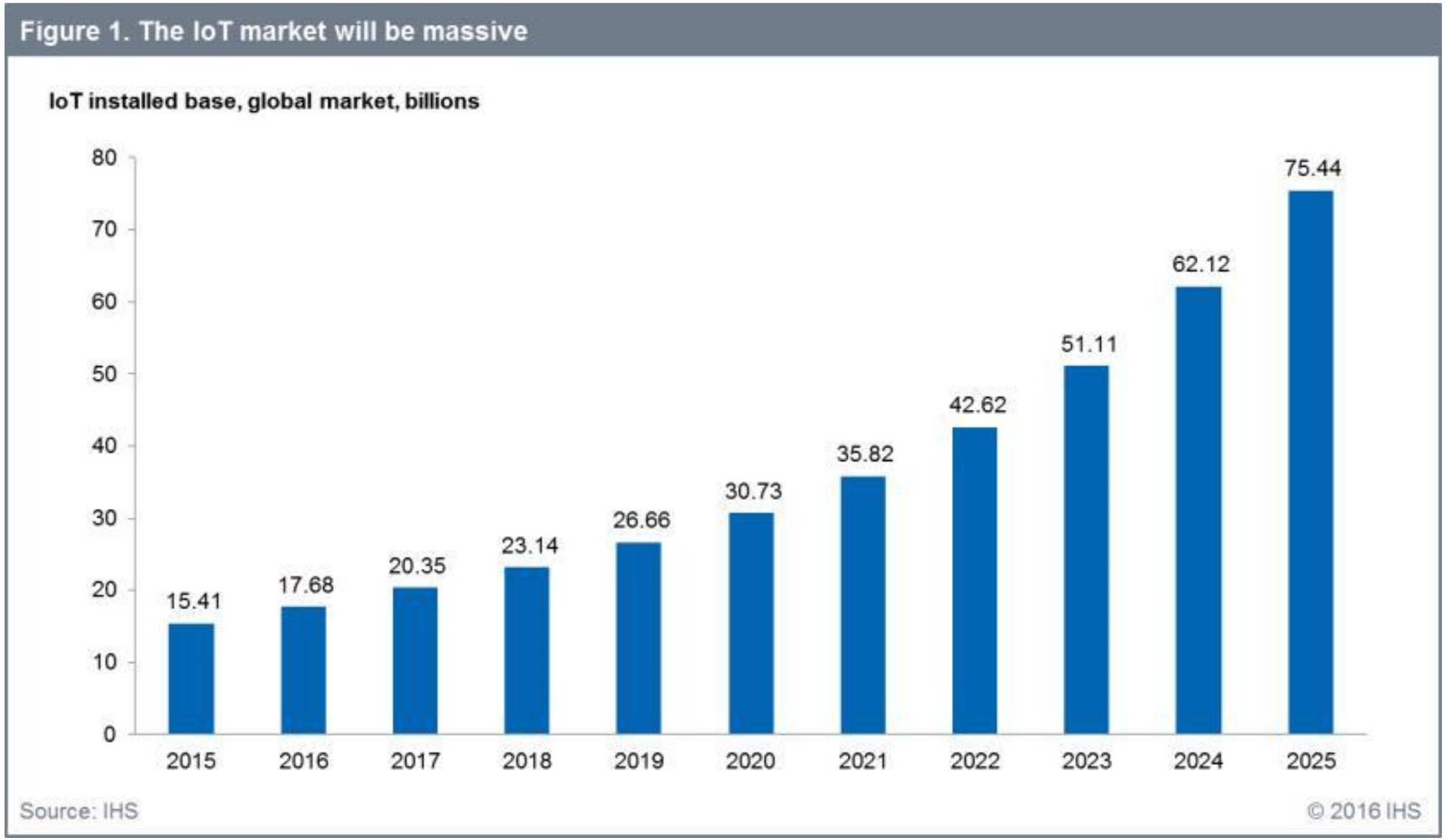
Department of Electrical Engineering and Computer Engineering

California State University, Los Angeles

cliu@calstatela.edu

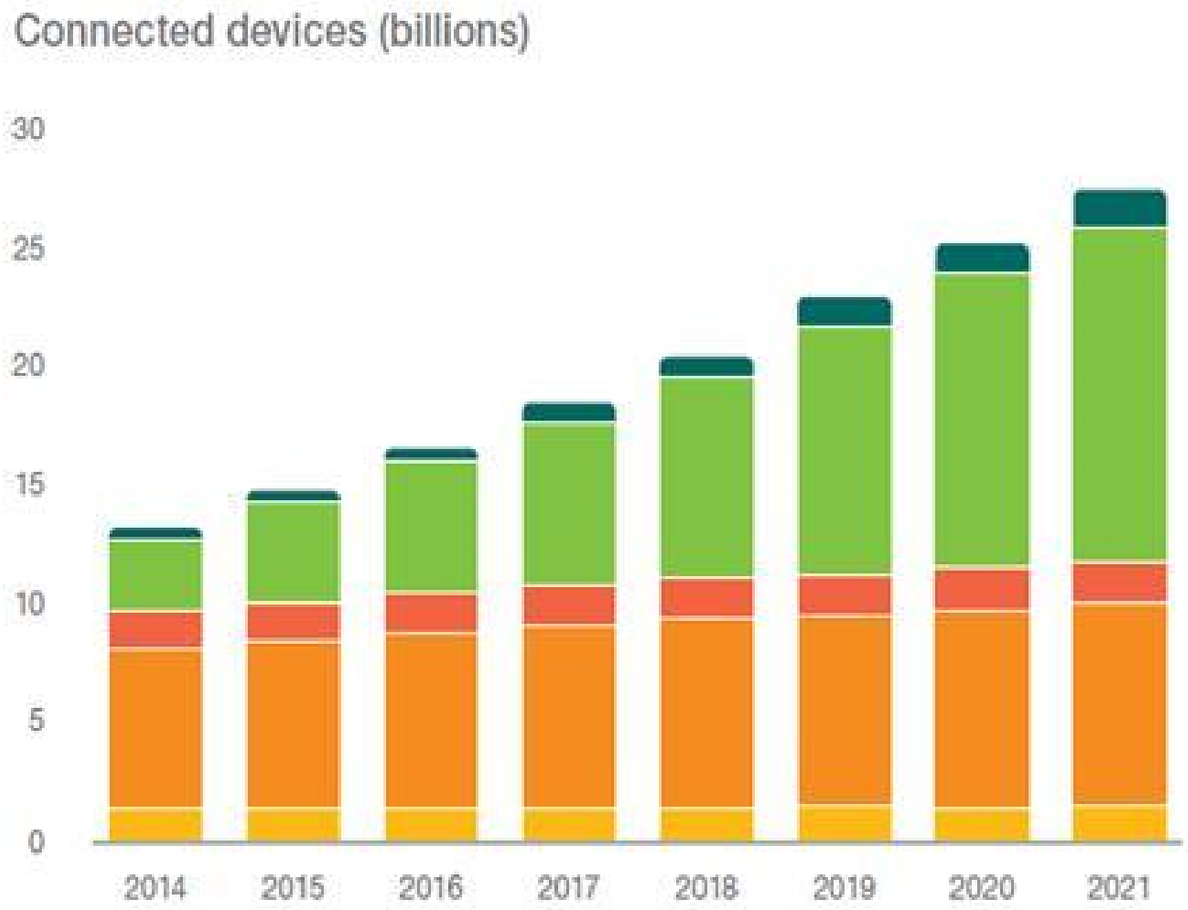


From <https://mobilunity.com/blog/iot-developer-salary-rates/>



From <https://blogs-images.forbes.com/louiscolumbus/files/2016/11/IHS.jpg>

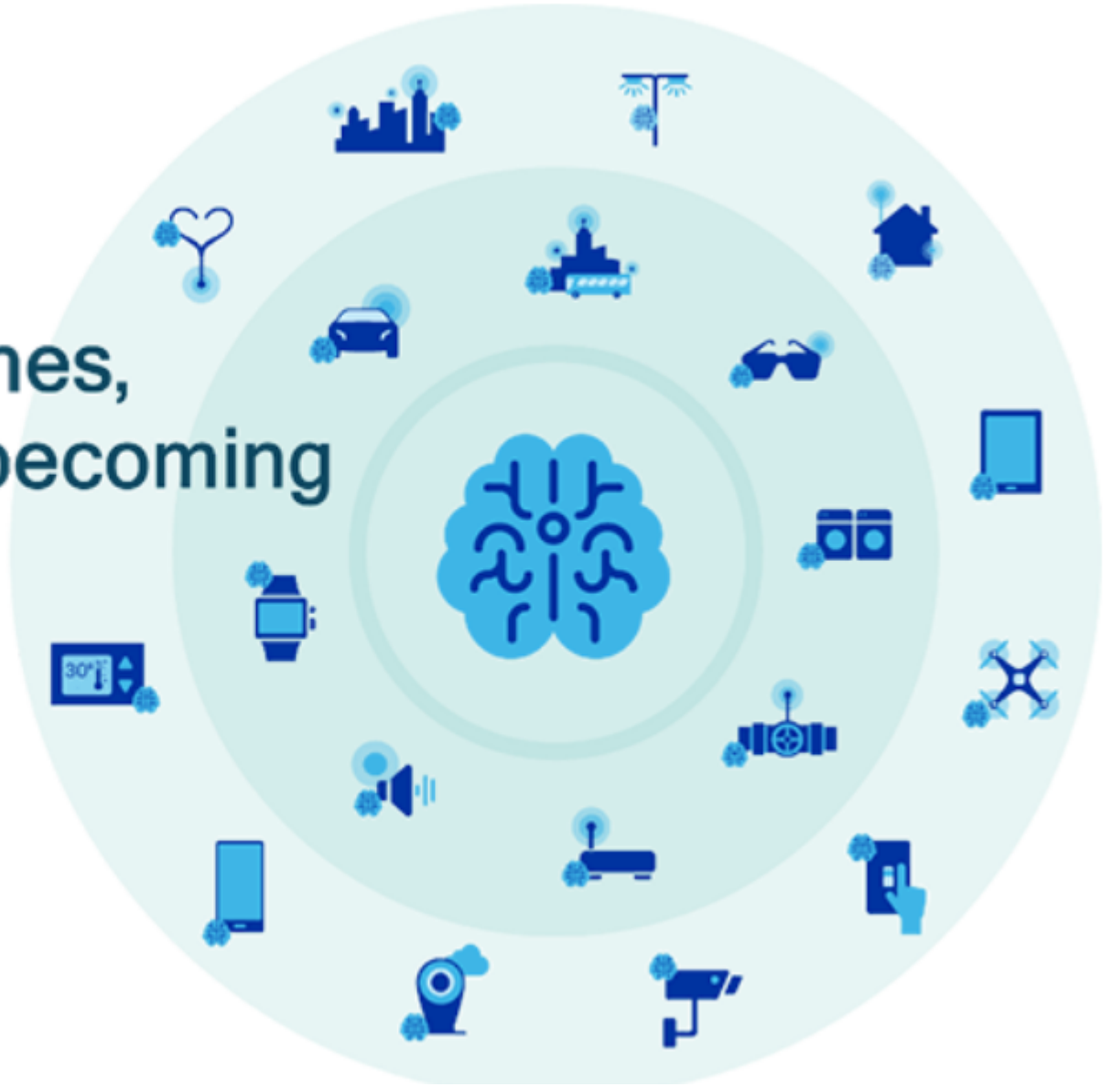
THE INTERNET OF THINGS



	15 billion	28 billion	CAGR 2015–2021
Cellular IoT	0.4	1.5	27%
Non-cellular IoT	4.2	14.2	22%
PC/laptop/tablet	1.7	1.8	1%
Mobile phones	7.1	8.6	3%
Fixed phones	1.3	1.4	0%
	2015	2021	

From <https://blogs-images.forbes.com/louiscolumbus/files/2016/07/Internet-of-Things-Forecast.jpg>

Devices, machines, and things are becoming more intelligent

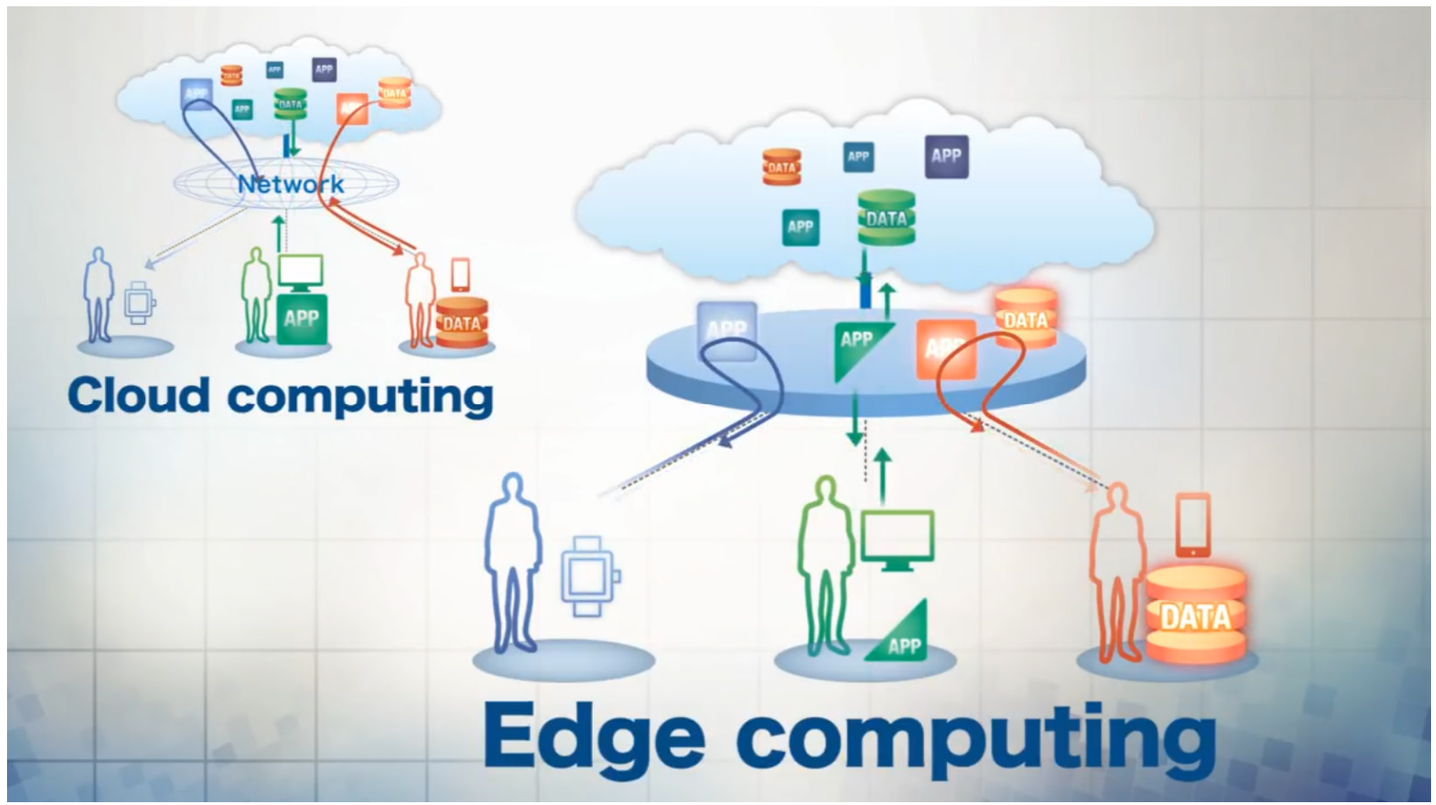


From < https://www.qualcomm.com/news/onq/2017/08/16/we-are-making-device-ai-ubiquitous >

Edge Computing

The ability to do advanced on-device processing and analytics is referred to as “edge computing.”
Edge computing is a counterpart to the cloud computing

Edge computing provides new possibilities in IoT applications,
machine learning for tasks such as object detection, face recognition, language processing, and obstacle avoidance



Instead of sending streams of images/videos to the cloud for processing, in-situ pre-processing is performed

Advantages: Saving in network and computing resources, reducing latency, improving security and privacy (personally identifiable information vs. demographic information)

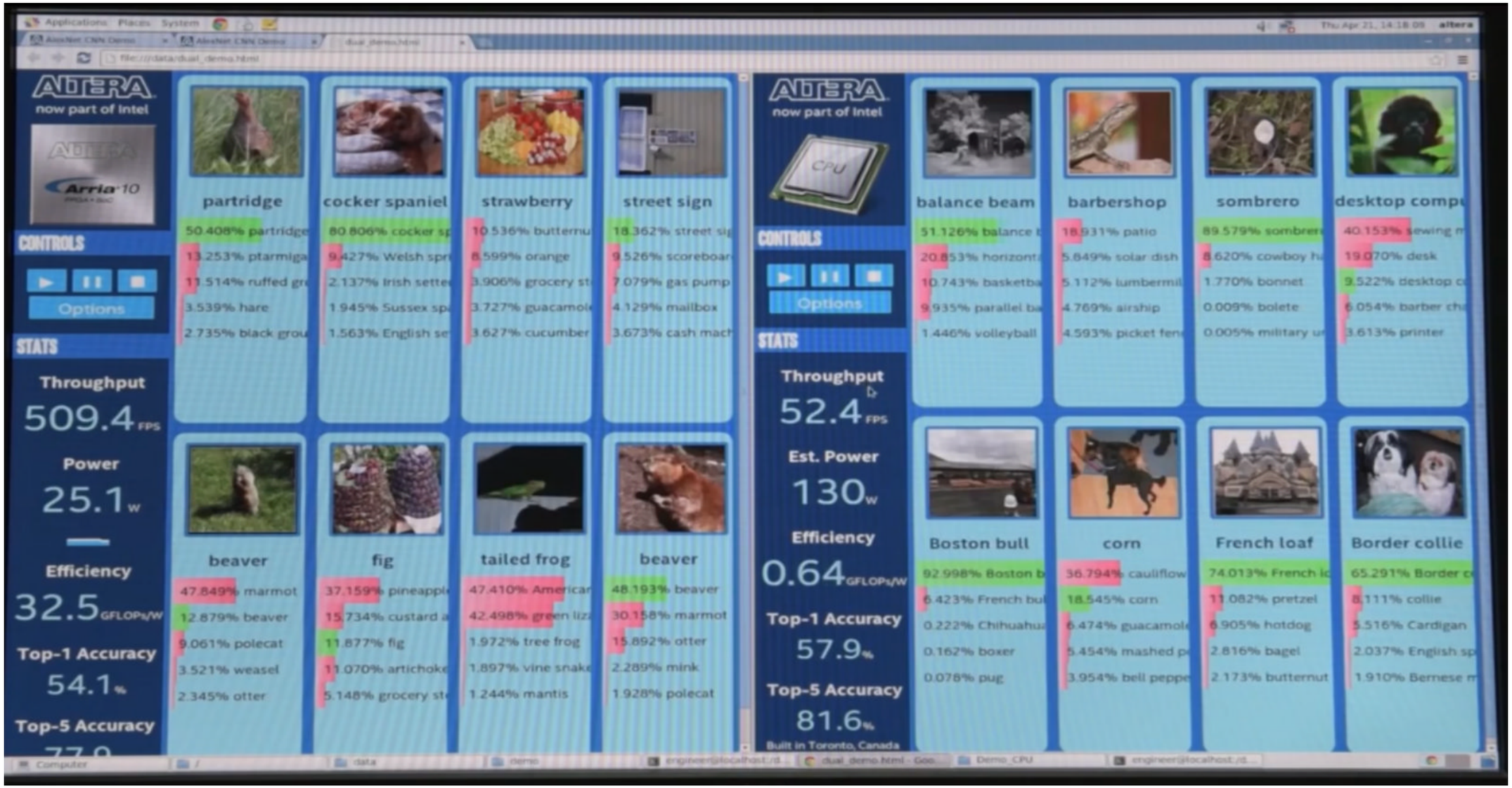
[E.g.] Proactive in-car service - natural language interface using Edge computing allows smart speakers to react more quickly by interpreting voice instructions locally.

Heterogeneous computer architectures are adopted for edge computing - integrating diverse engines such as CPUs, GPUs and DSPs — in IoT devices so that different workloads are assigned to the most efficient compute engine, thus improving performance and power efficiency.

[E.g.] The Hexagon DSP with Qualcomm Hexagon Vector eXtensions on Snapdragon 835 has been shown to offer a 25X improvement in energy efficiency and an 8X improvement in performance when compared against running the same workloads (GoogleNet Inception Network) on the Qualcomm Kryo CPU.

From <https://www.qualcomm.com/news/ong/2017/08/16/we-are-making-device-ai-ubiquitous>

[Convolutional Neural Network CNN Implementation on Altera FPGA using OpenCL]



https://www.youtube.com/watch?v=78Qd5t-Mn0s

Heterogeneous Computer Architectures

- CPUs
- GPUs
- Vector Processors
- Image/Signal Processors
- FPGAs

Suppose you want to add two vectors of numbers.
There are many ways to spell this – programming paradigms.

C uses a loop spelling

```
for(i=0;i<n;++i) a[i]=b[i]+c[i];
```

Matlab uses a vector spelling

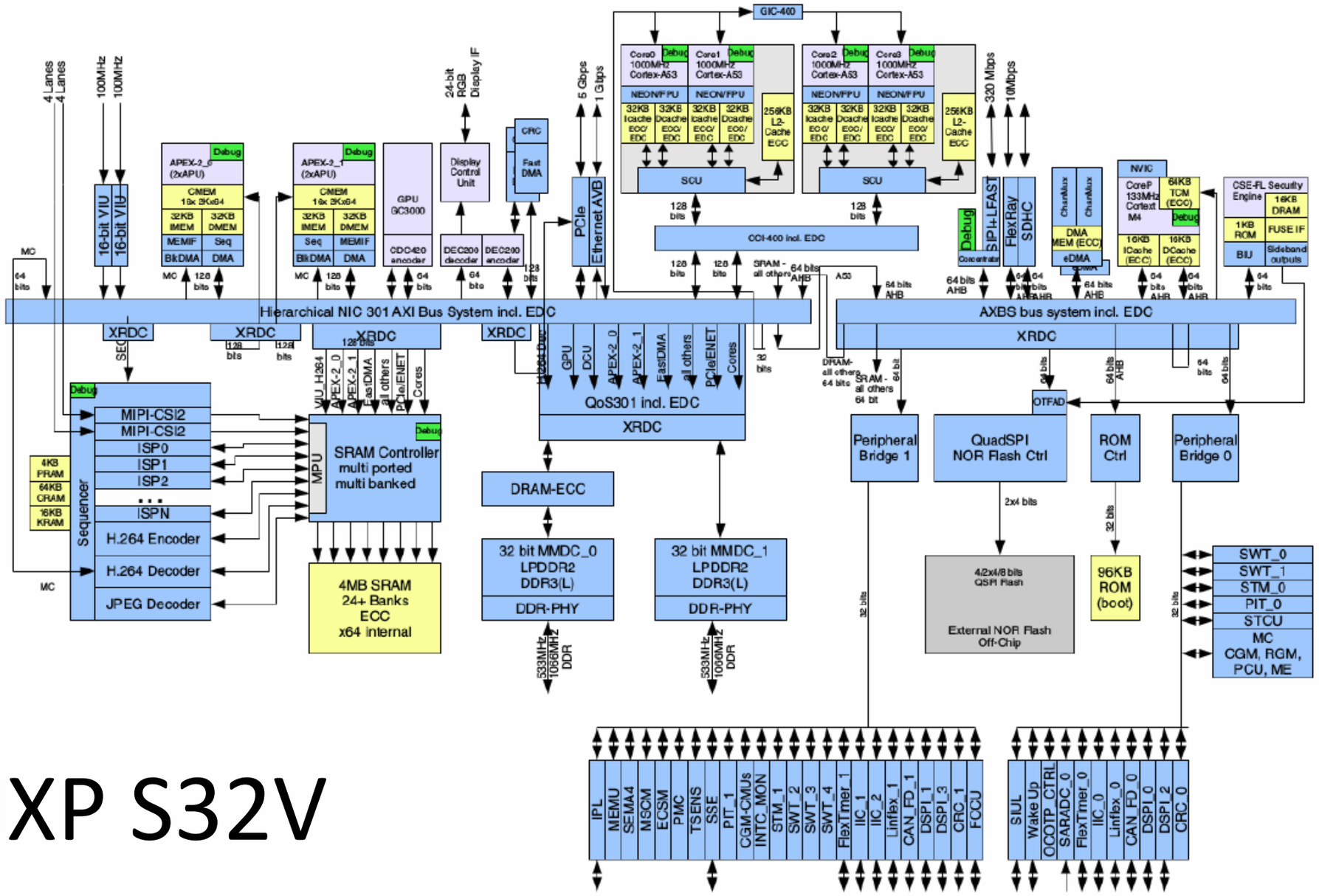
```
a=b+c;
```

SIMD uses a "short vector" spelling

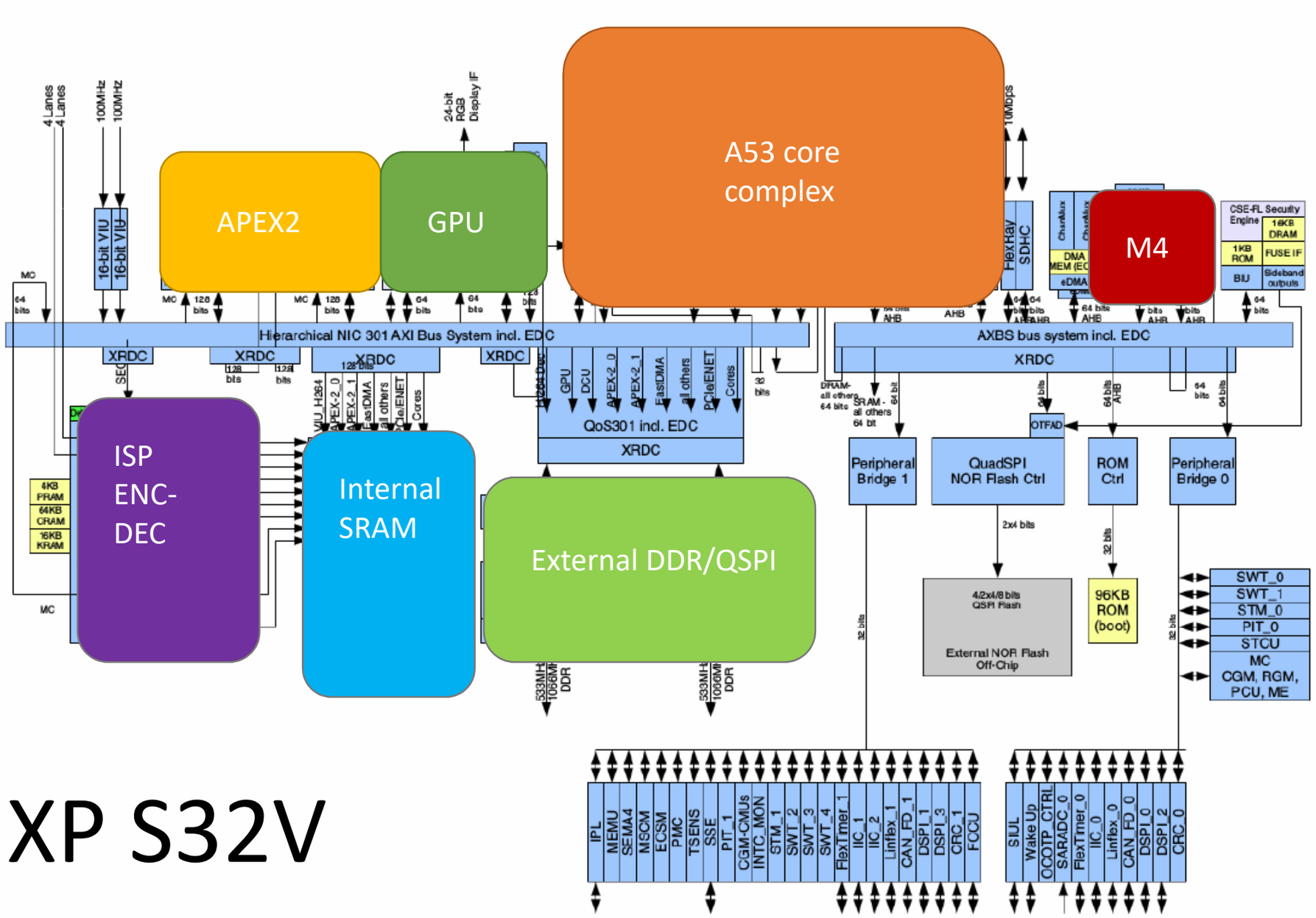
```
void add(uint32_t *a, uint32_t *b, uint32_t *c,
int n) {

   for(int i=0; i<n; i+=4) {
   //compute c[i], c[i+1], c[i+2], c[i+3]

         uint32x4_t b4 = vld1q_u32(b+i);
         uint32x4_t c4 = vld1q_u32(c+i);
         uint32x4_t a4 = vaddq_u32(b4,c4);
         vst1q_u32(a+i,a4);
   }
}
```

SIMT uses a "scalar" spelling

```
__global__ void
add(float *a, float *b, float *c) {
  int i = blockIdx.x * blockDim.x +
          threadIdx.x;

a[i]=b[i]+c[i]; //no loop!
}
```


GIC-400
Core0 1000MHz Cortex-A53
Core1 1000MHz Cortex-A53
Core2 1000MHz Cortex-A53
Core3 1000MHz Cortex-A53
NEON/FPU
32KB Icache ECC/EDC
32KB Dcache ECC/EDC
256KB L2-Cache ECC
SCU
CCI-400 incl. EDC
Hierarchical NIC 301 AXI Bus System incl. EDC
AXBS bus system incl. EDC
XRDC
QoS301 incl. EDC
APEX-2_0 (2xAPU)
APEX-2_1 (2xAPU)
CMEM 16x 2Kx64
32KB IMEM
32KB DMEM
MEMIF
Seq
BlkDMA
DMA
GPU GC3000
GDC400 encoder
Display Control Unit
Fast DMA
DEC200 decoder
DEC200 encoder
PCIe
Ethernet AVB
16-bit VIU
MIPI-CSI2
ISP0
ISP1
ISP2
ISPN
H.264 Encoder
H.264 Decoder
JPEG Decoder
Sequencer
4KB PRAM
64KB CRAM
16KB KRAM
SRAM Controller multi ported multi banked
MPU
4MB SRAM 24+ Banks ECC x64 internal
DRAM-ECC
32 bit MMDC_0 LPDDR2 DDR3(L)
32 bit MMDC_1 LPDDR2 DDR3(L)
DDR-PHY
533MHz 1066MHz DDR
Peripheral Bridge 1
Peripheral Bridge 0
QuadSPI NOR Flash Ctrl
OTFAD
ROM Ctrl
4/2x4/8 bits QSPI Flash
External NOR Flash Off-Chip
96KB ROM (boot)
SPI+LFAST
FlexRay
SDHC
DMA MEM (ECC)
eDMA
INTC
CoreP 133MHz Cortex M4
64KB TCM (ECC)
16KB ICache (ECC)
16KB DCache (ECC)
CSE-FL Security Engine
16KB DRAM
1KB ROM
FUSE IF
BIU
Sideband outputs
SWT_0
SWT_1
STM_0
PIT_0
STCU
MC CGM, RGM, PCU, ME
IPL
MEMU
SEMA4
MSCM
ECSM
PMC
TSENS
SSE
PIT_1
CGM-CMUs
INTC_MON
STM_1
SWT_2
SWT_3
SWT_4
FlexTimer_1
IIC_1
IIC_2
Linflex_1
CAN_FD_1
DSPI_3
CRC_1
FCCU
SIUL
Wake Up
OCOTP_CTRL
SARADC_0
FlexTimer_0
IIC_0
Linflex_0
CAN_FD_0
DSPI_0
DSPI_2
CRC_0

NXP S32V


A53 core complex
APEX2
GPU
M4
ISP ENC-DEC
Internal SRAM
External DDR/QSPI
Hierarchical NIC 301 AXI Bus System incl. EDC
AXBS bus system incl. EDC
QoS301 incl. EDC
XRDC
Peripheral Bridge 1
Peripheral Bridge 0
QuadSPI NOR Flash Ctrl
ROM Ctrl
96KB ROM (boot)
External NOR Flash Off-Chip
SWT_0
SWT_1
STM_0
PIT_0
STCU
MC CGM, RGM, PCU, ME

NXP S32V

Each engine has a best efficiency on certain type of functions. To let the complete system working at highest efficiency each engine needs to work in parallel in pipeline mode.

Image sensor → Processing and Conditioning → Filtering → Edge Extraction → Transformation Feature Classification → Info Extraction → Other algos

GPU → Display

Typical ISP functions

Black Level and Dead Pixel processing
Black level, Vignetting
Exposure control, color balance analysis
Geometric distortion corrections, chromatic aberration
HDR
De-mosaic Bayer pattern
RGB->YUV420, channel gain
Spatial de-noise
Gamma correction
Image Scaling

Typical *APEX* functions

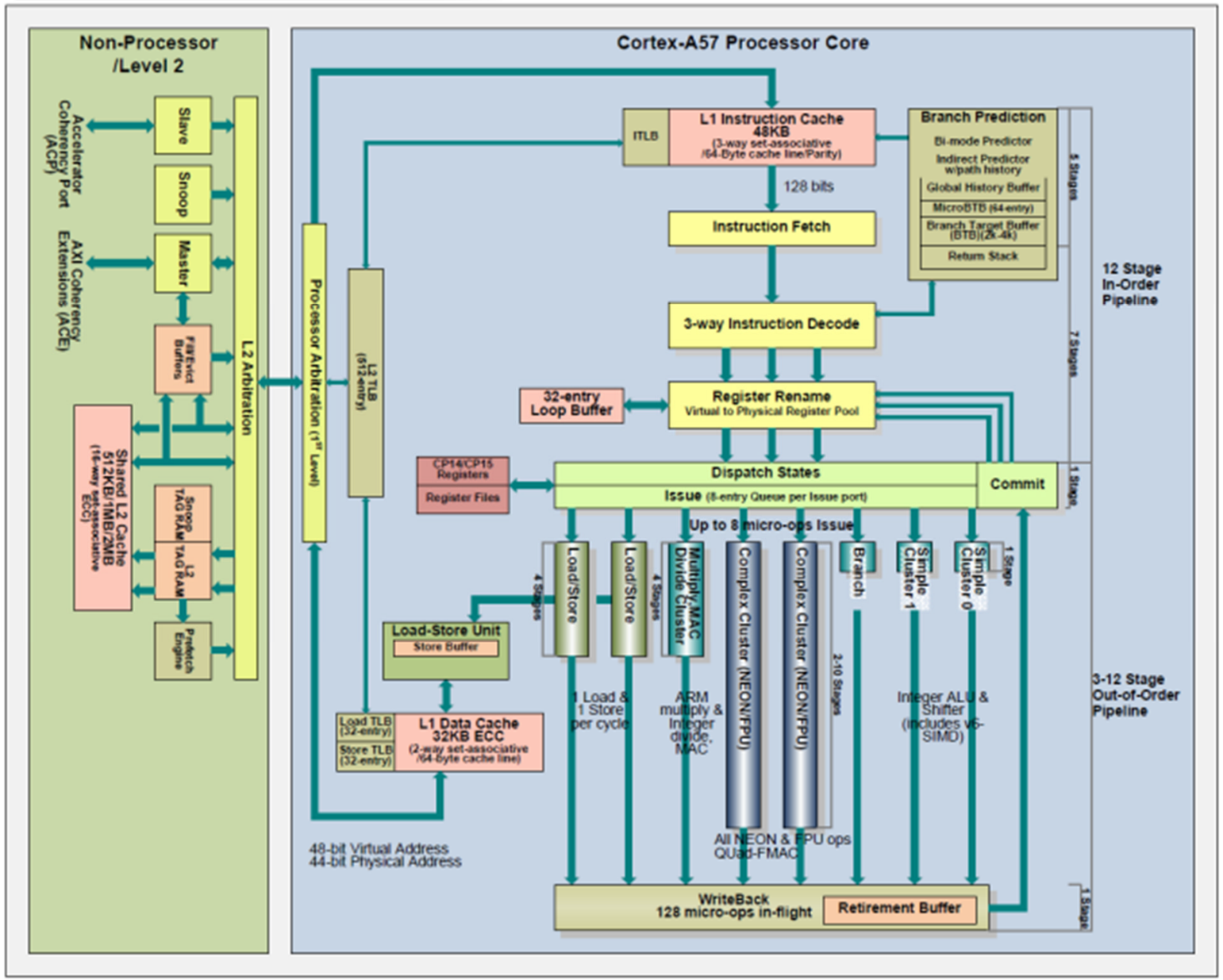
Whatever Kernel based filter:
- Sobel X
- Median
- Histogram
- Integral Image
- FAST
- BRIEF
- ARB
- FIR
- Kirsh
- Laplace filter
- Gaussian
- RGB to YUV
- YUV 2 RGB
- YUV 420 to RGB565
- Harris Corner
- Shi–Tomasi Corner
- Lens Correction

ARM53 Neon

APEX2

ARM53 Neon

ARM Cortex-A57 Block Diagram



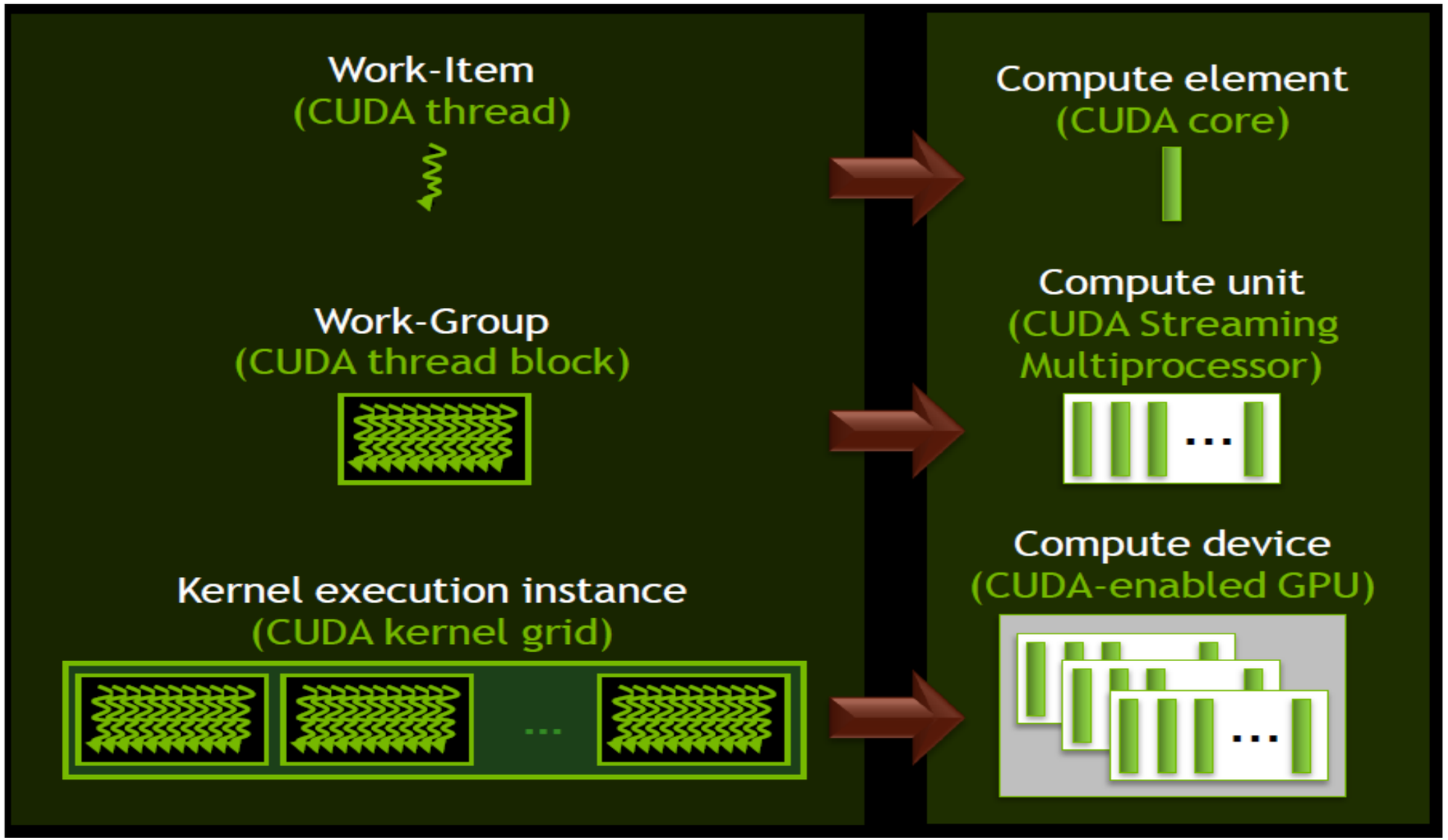
Programming Paradigms

	OpenACC	OpenMP	OpenCL	CUDA
Parallelism	- data parallelism - asynchronous task parallelism - device only	- data parallelism - asynchronous task parallelism - host and device	- data parallelism - asynchronous task parallelism - host and device	- data parallelism - asynchronous task parallelism - device only
Architecture abstraction	- memory hierarchy - explicit data mapping and movement	- memory hierarchy - data and computation binding - explicit data mapping and movement	- memory hierarchy - explicit data mapping and movement	- memory hierarchy - explicit data mapping and movement
Synchronization	- reduction - join	- barrier - reduction - join	- barrier; - reduction	- barrier
Framework implementation	compiler directives for C/C++ and Fortran	compiler directives for C/C++ and Fortran	C/C++ extension	C/C++ extension

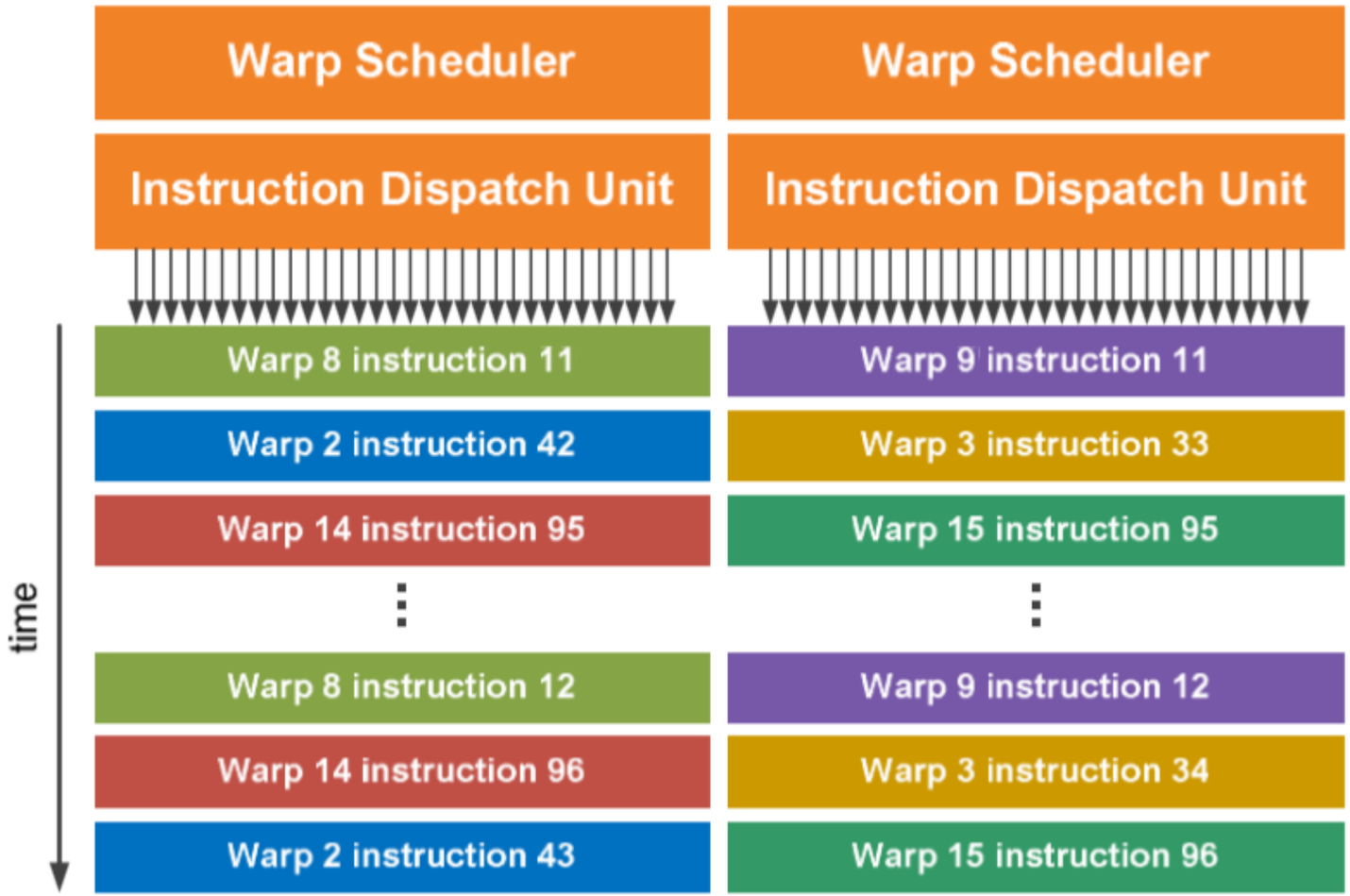
NVIDIA GUP: GeForce_GTX_480_Fermi - Single Instruction Multiple Thread (SIMT) architecture

https://www.techpowerup.com/reviews/NVIDIA/GeForce_GTX_480_Fermi/

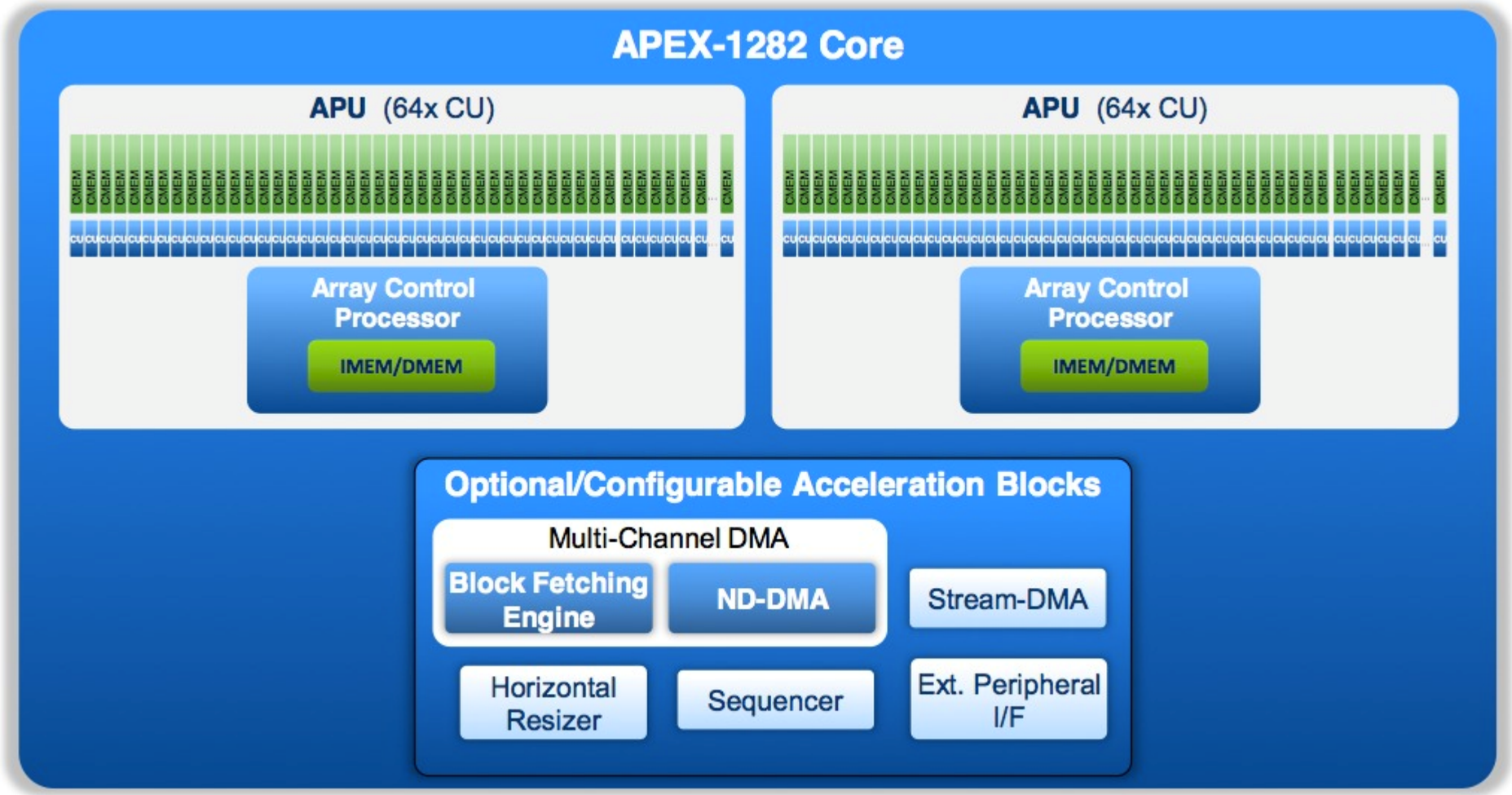
CUDA and OpenCL terminology correspondence.



The SMs schedule and execute threads in lockstep groups of 32 threads called *warps*.



CogniVue's APEX vision processor core architectures



https://www.bdti.com/InsideDSP/2015/06/30/CogniVue

Example vector operator -

vec16s operator- (vec16s va,
vec16s vb
)

Vector Subtraction.

Parameters

va Source vector

vb Source vector

Returns

(va-vb)

```
vec16s a = (vec16s) src[0];
vec16s b = (vec16s) src[1];

vec16s res = a-b;
```

	CU0	CU1	CU2	CU3	CU4	CU5	CU6	CU7	CU8	CU9	CU10	CU11	CU12	CU13	CU14	CU15	CU16	CU17	CU18	CU19	CU20	CU21	CU22	CU23	CU24	CU25	CU26	CU27	CU28	CU29	CU30	CU31	
0	C9	9E	9C	AC	6B	6C	6C	77	85	82	85	85	86	87	84	85	8A	82	83	7D	6A	9F	9C	99	9B	9F	D7	6E	7B	7C	7C	73	a
1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
2																																	
3	A3	9D	9D	AB	5D	6A	6B	7A	84	80	86	85	87	86	85	86	87	81	83	7A	74	A2	9F	99	9D	B2	C7	77	7A	7D	7D	6D	b
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
5																																	
6	6C	3B	39	57	C8	D6	D7	F1	09	02	0B	0A	0D	0D	09	0B	11	03	06	F7	DE	41	3B	32	38	51	9E	E5	F5	F9	F9	E0	res
7	01	01	01	01	00	00	00	00	01	01	01	01	01	01	01	01	01	01	01	00	00	01	01	01	01	01	01	00	00	00	00	00	

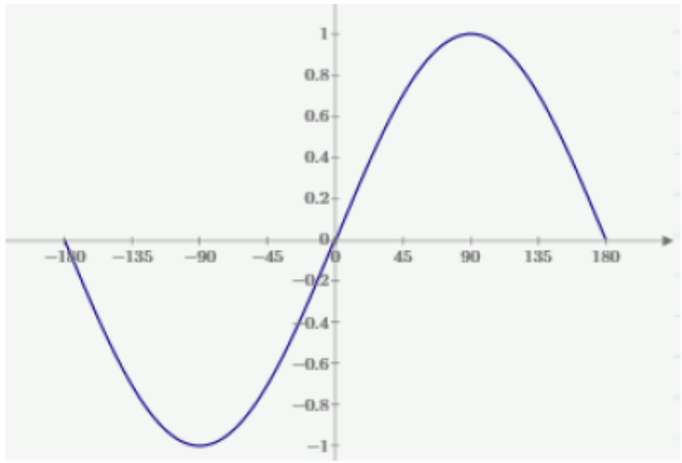
Understand vif

```
vec08u V0,V1;
vec16u VR;
vif (V0 < V1)          // condition
{
   VR = V0 + V1;     // TRUE
}
velse
{
   VR = V0;          // FALSE
}
vendif
```

CMEM final after executing the velse statement and the second predicated instruction.

CU0	CU1	CU2	CU3	CU4	CU5	CU6	CU7	CU8	CU9	CU10	CU11	CU12	CU13	CU14	CU15	CU16	CU17	CU18	CU19	CU20	CU21	CU22	CU23	CU24	CU25	CU26	CU27	CU28	CU29	CU30	CU31	
0	0	1	0	0	0	0	1	0	0	1	0	1	0	1	1	0	0	0	0	1	1	1	0	1	1	0	1	0	1	1	0	register
CU0	CU1	CU2	CU3	CU4	CU5	CU6	CU7	CU8	CU9	CU10	CU11	CU12	CU13	CU14	CU15	CU16	CU17	CU18	CU19	CU20	CU21	CU22	CU23	CU24	CU25	CU26	CU27	CU28	CU29	CU30	CU31	
C9	9E	9C	AC	6B	6C	6C	77	85	82	85	85	86	87	84	85	8A	82	83	7D	6A	9F	9C	99	9B	9F	D7	6E	7B	7C	7C	73	V0
A3	9D	9D	AB	5D	6A	6B	7A	84	80	86	85	87	86	85	86	87	81	83	7A	74	A2	9F	99	9D	B2	C7	77	7A	7D	7D	6D	V1
C9	9E	39	AC	6B	6C	6C	F1	85	82	0B	85	0D	87	09	0B	8A	82	83	7D	DE	41	3B	99	38	51	D7	E5	7B	F9	F9	73	VR
00	00	01	00	00	00	00	00	00	00	01	00	01	00	01	01	00	00	00	00	00	01	01	00	01	01	00	00	00	00	00	00	

vload example; vsin



Typical fast sin can be build with a look-up table with 361 elements and use the angle as the table index

We can think to build the vector vsin in the same way.

v_angle is now a vector and vsin(v_angle) returns a vector with

res{i} = sin(v_angle{i})

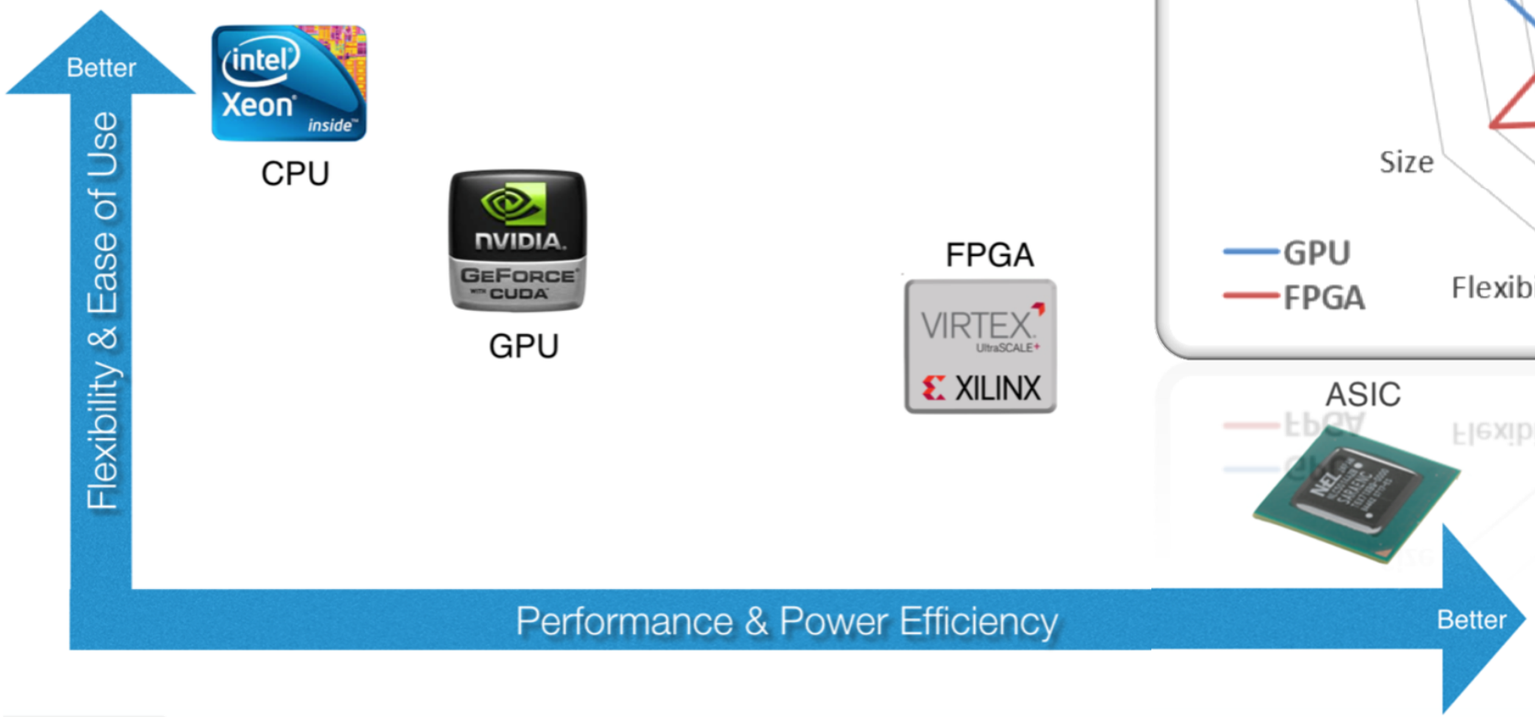
```
 .data global 361 VSIN_t VMw
0x0000  0x0000  0x0000  0x0000          …… //sin(-180) * 32768
0xFEE3 0xFEE3 0xFEE3 0xFEE3          ….   //sin(-179) * 32768
0xFDC5 0xFDC5 0xFDC5 0xFDC5          ….   //sin(-178) * 32768
……
0xFDC5 0xFDC5 0xFDC5 0xFDC5          ….   //sin(-178) * 32768
0xFEE3 0xFEE3 0xFEE3 0xFEE3            ….   //sin(-1) * 32768
0x0000  0x0000  0x0000  0x0000          …… //sin(0) * 32768
0x011D 0x011D 0x011D 0x011D            …… //sin(1) * 32768
0x023B 0x023B 0x023B 0x023B              …… //sin(2) * 32768
……
0x011D 0x011D 0x011D 0x011D            …… //sin(179) * 32768
0x0000  0x0000  0x0000  0x0000          …… //sin(180) * 32768
```

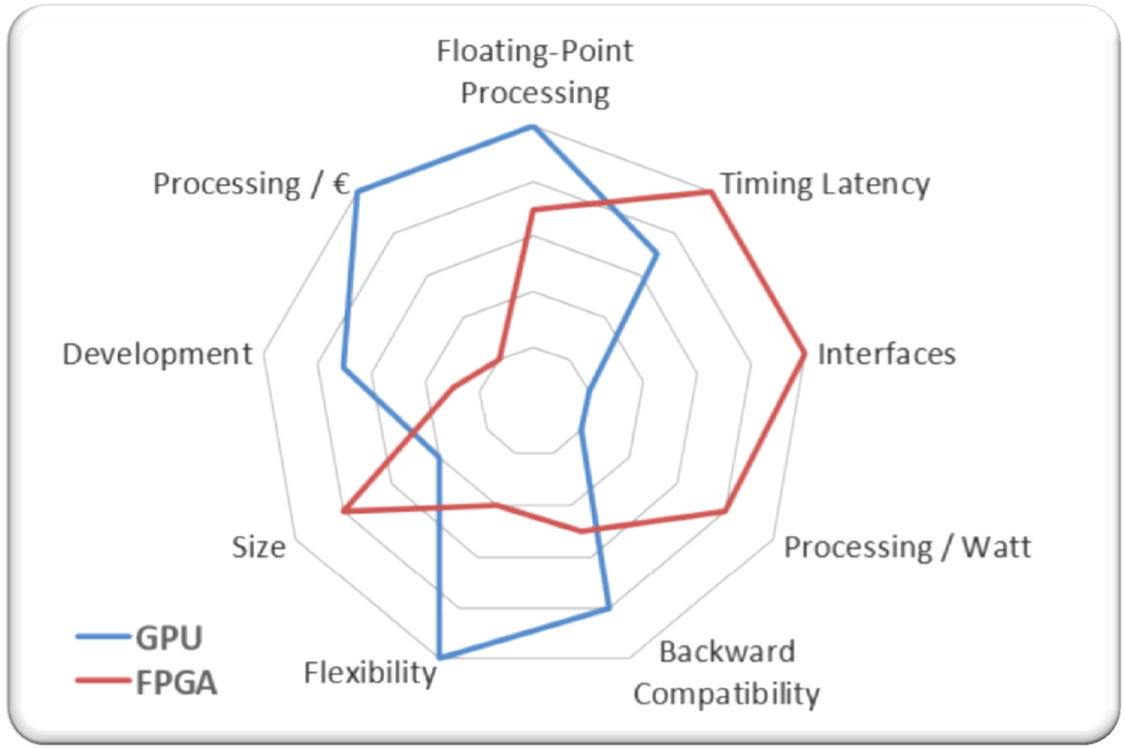
```
vec16s VSIN_t[361];
const vec16s *VSIN =&VSIN_t[0];

vec16s v_sin(vec16s v_angle)
{
      return  vload(VSIN, v_angle);
}
```

$V\_res_{28} = sin(v\_angle_{28}) = sin(-100) = 0x81F2$ that needs to be (>>15)

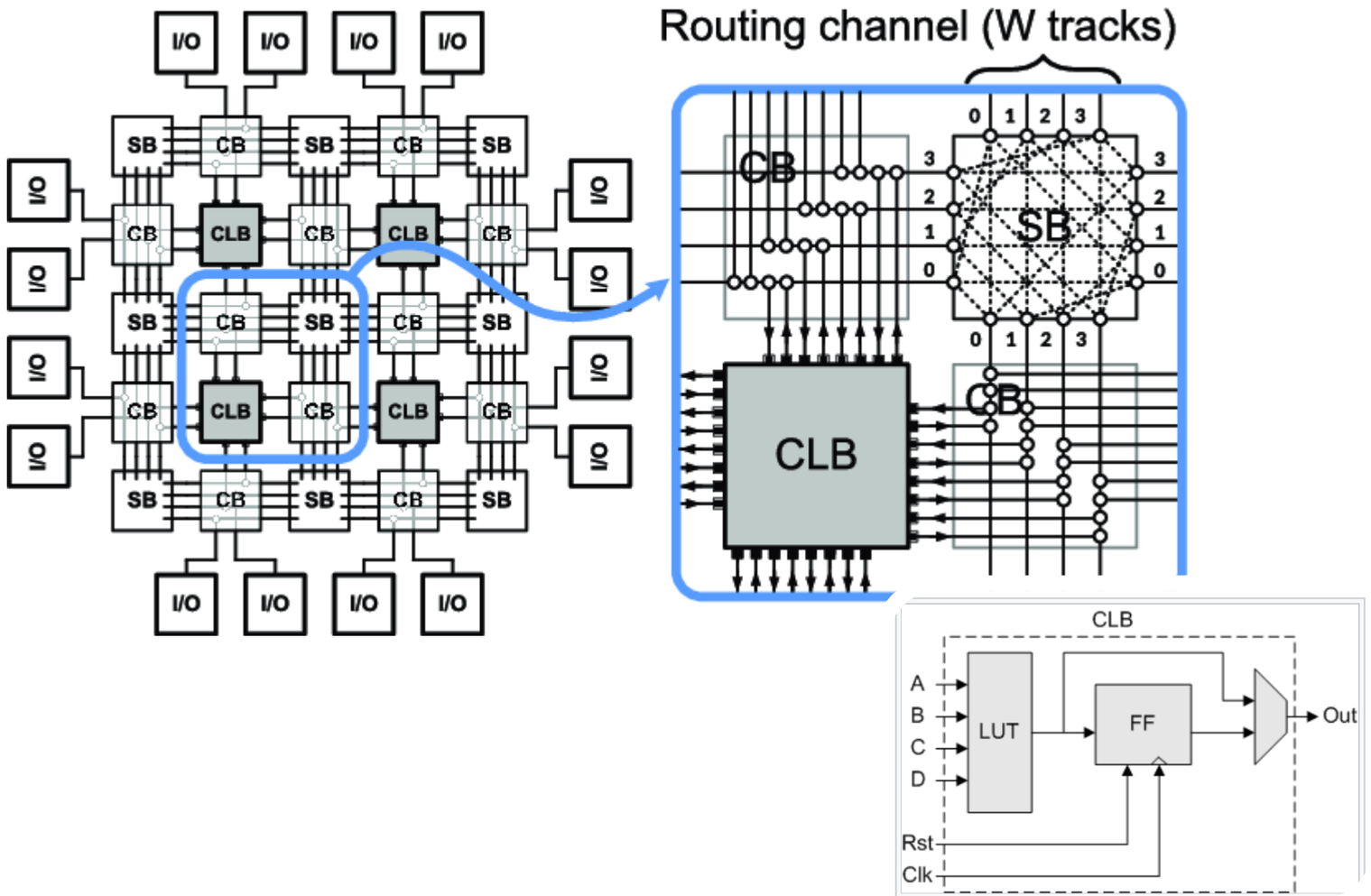
CU0	CU1	CU2	CU3	CU4	CU5	CU6	CU7	CU8	CU9	CU10	CU11	CU12	CU13	CU14	CU15	CU16	CU17	CU18	CU19	CU20	CU21	CU22	CU23	CU24	CU25	CU26	CU27	CU28	CU29	CU30	CU31	
0	10	20	30	40	50	60	70	80	90	-10	-20	-30	-40	-50	-60	-70	-80	-90	100	110	120	130	140	150	160	170	180	-100	-90	-80	-70	v_angle
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00	3A	C7	FF	46	0D	D9	47	0E	00	C6	39	01	BA	F3	27	B9	F2	00	0E	47	D9	0D	46	00	C7	3A	00	F2	00	F2	B9	v_sin(v_angle)
00	16	2B	3F	52	62	6E	78	7E	80	E9	D4	C0	AD	9D	91	87	81	80	7E	78	6E	62	52	40	2B	16	00	81	80	81	87	

Field programmable Gate Array (FPGA) is another Choice!



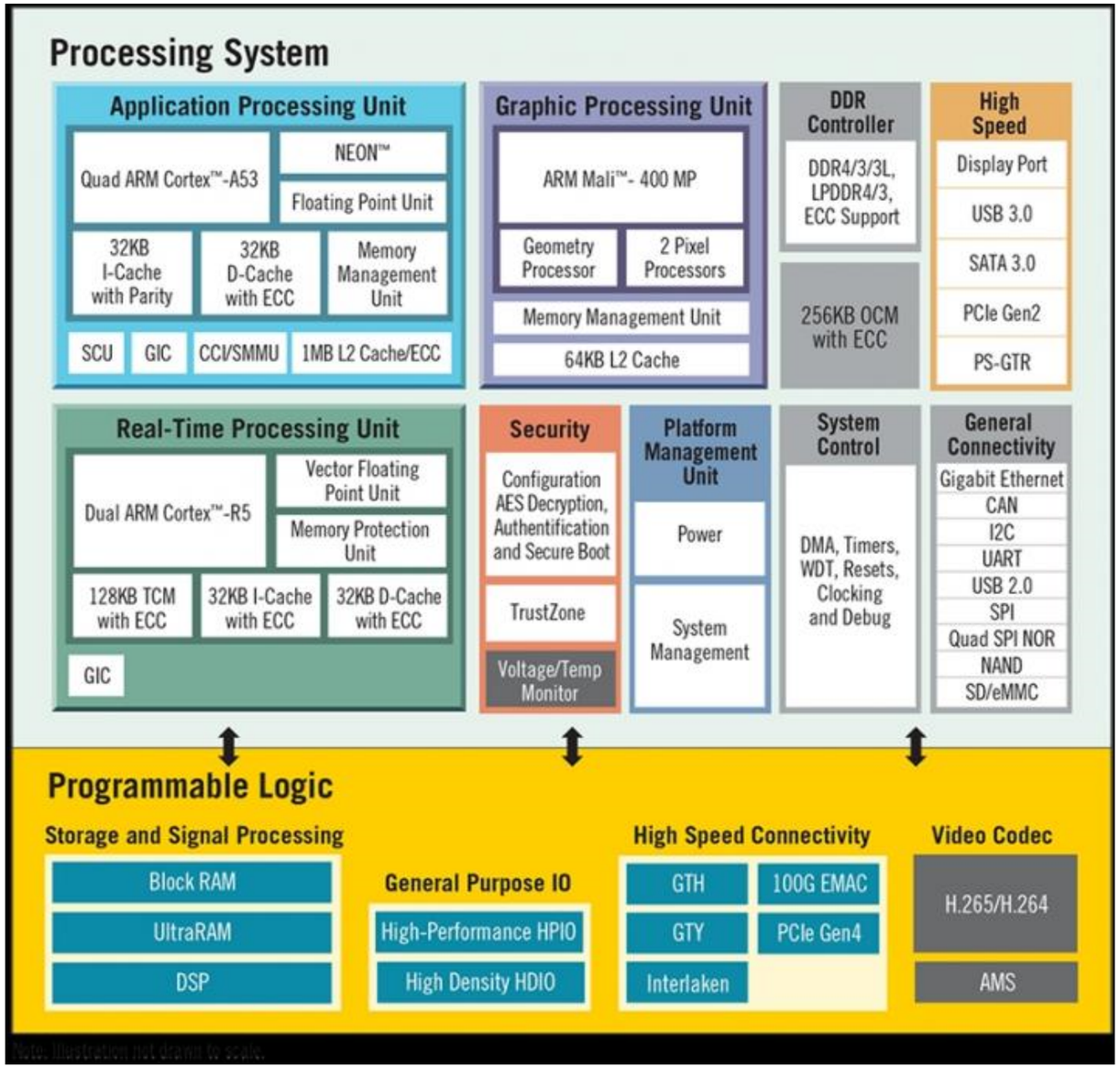


Island-style global FPGA architecture. A unit tile consists of Configurable Logic Block (CLB), Connect Box (CB) and Switch Box(SB).



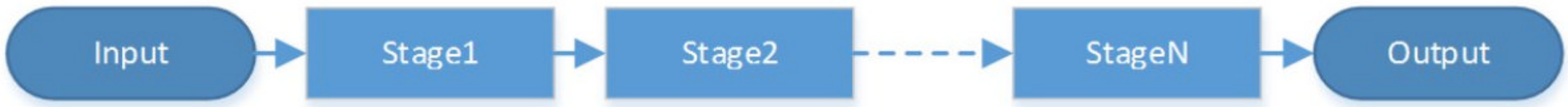
Heterogeneous MPSoC

Xilinx's 16nm UltraScale+



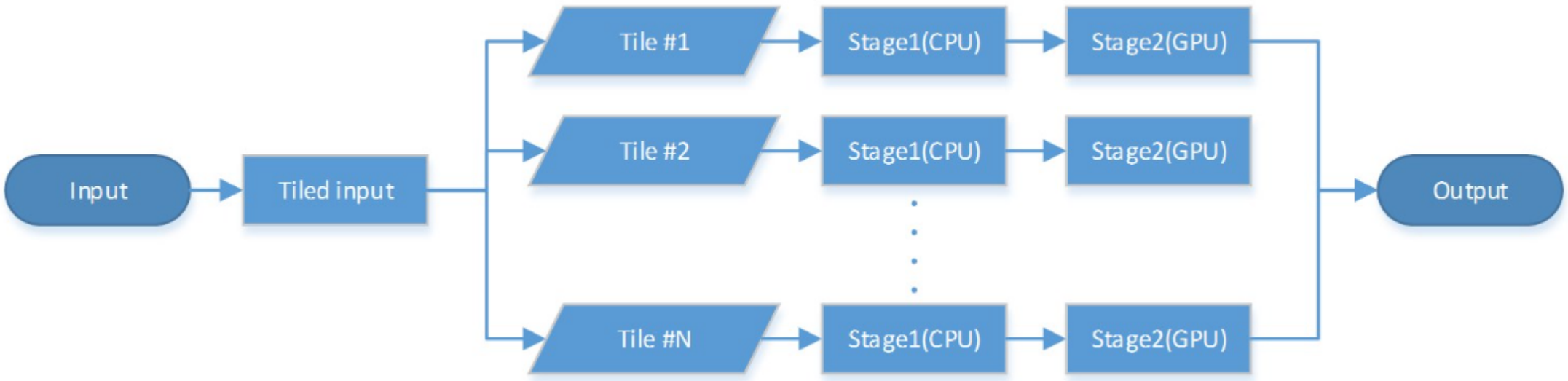
SoC Design Challenges

Data streaming

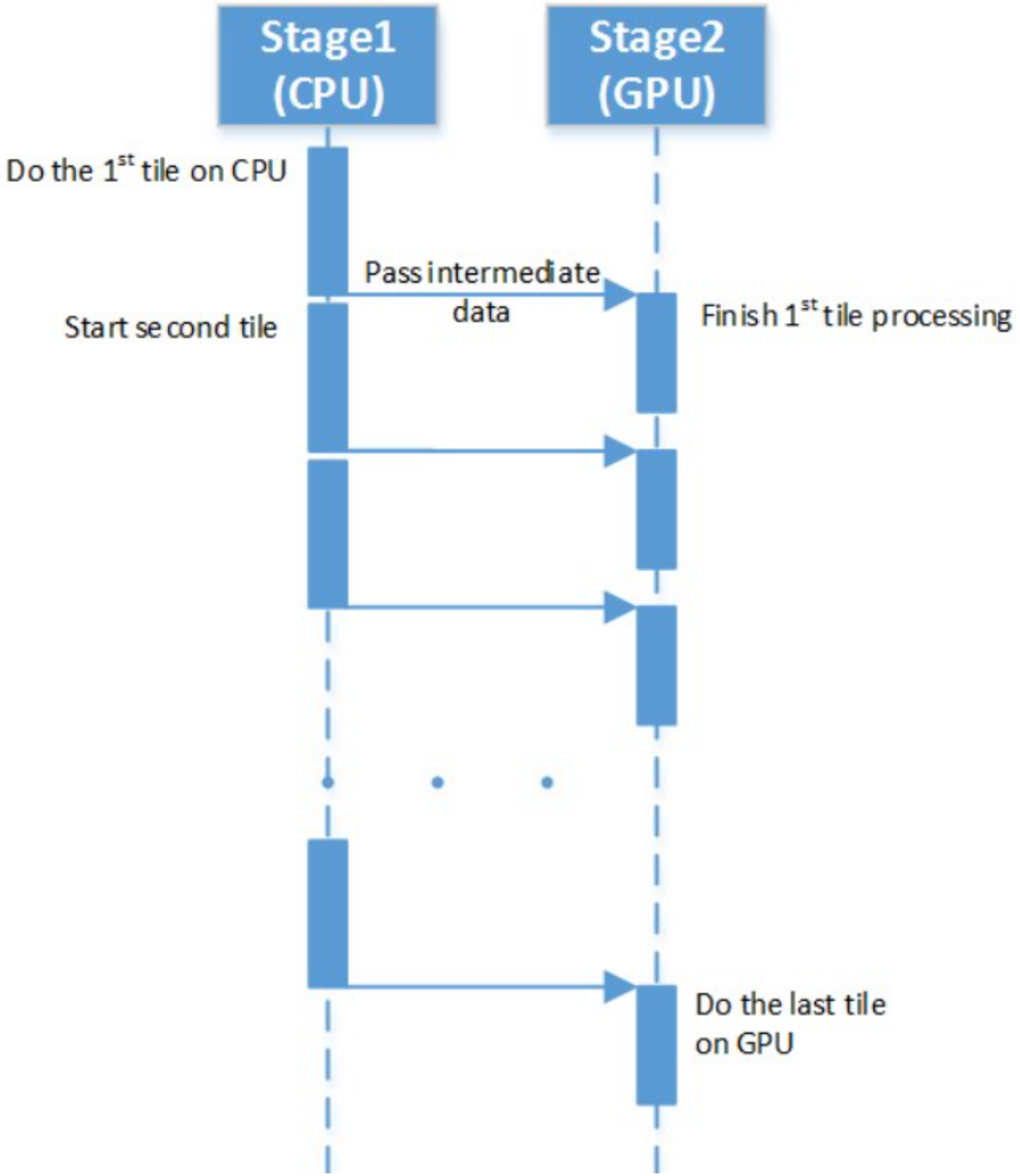


General algorithm execution model

Pipelining


Input
Tiled input
Tile #1
Tile #2
Tile #N
Stage1(CPU)
Stage1(CPU)
Stage1(CPU)
Stage2(GPU)
Stage2(GPU)
Stage2(GPU)
Output

Pipelining


Stage1
(CPU)
Stage2
(GPU)
Do the 1st tile on CPU
Pass intermediate
data
Start second tile
Finish 1st tile processing
Do the last tile
on GPU

Data Mapping and Remapping

- Coalescing global memory accesses to minimize number of memory transactions
- Improving memory locality of next-stage thread access
- Improving memory locality of inter-thread accesses

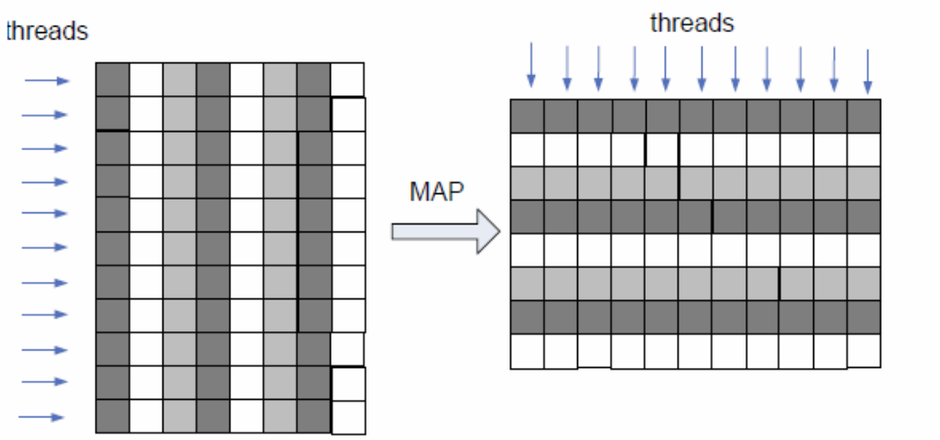


Figure 6: Row-major to column-major transformation.

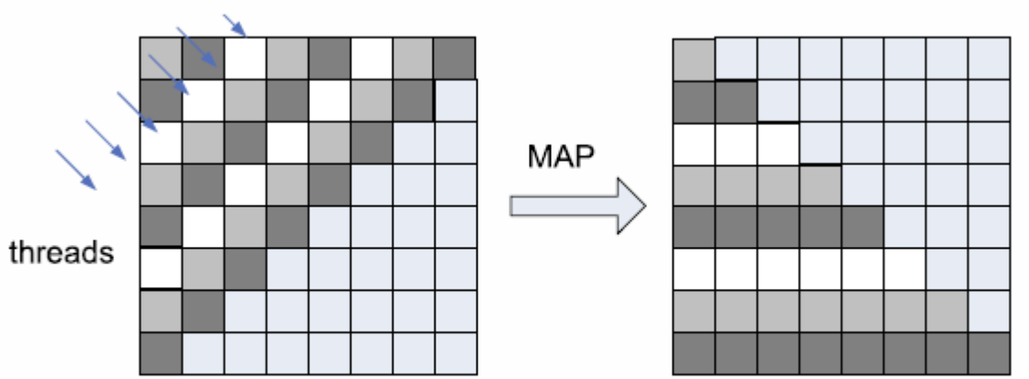
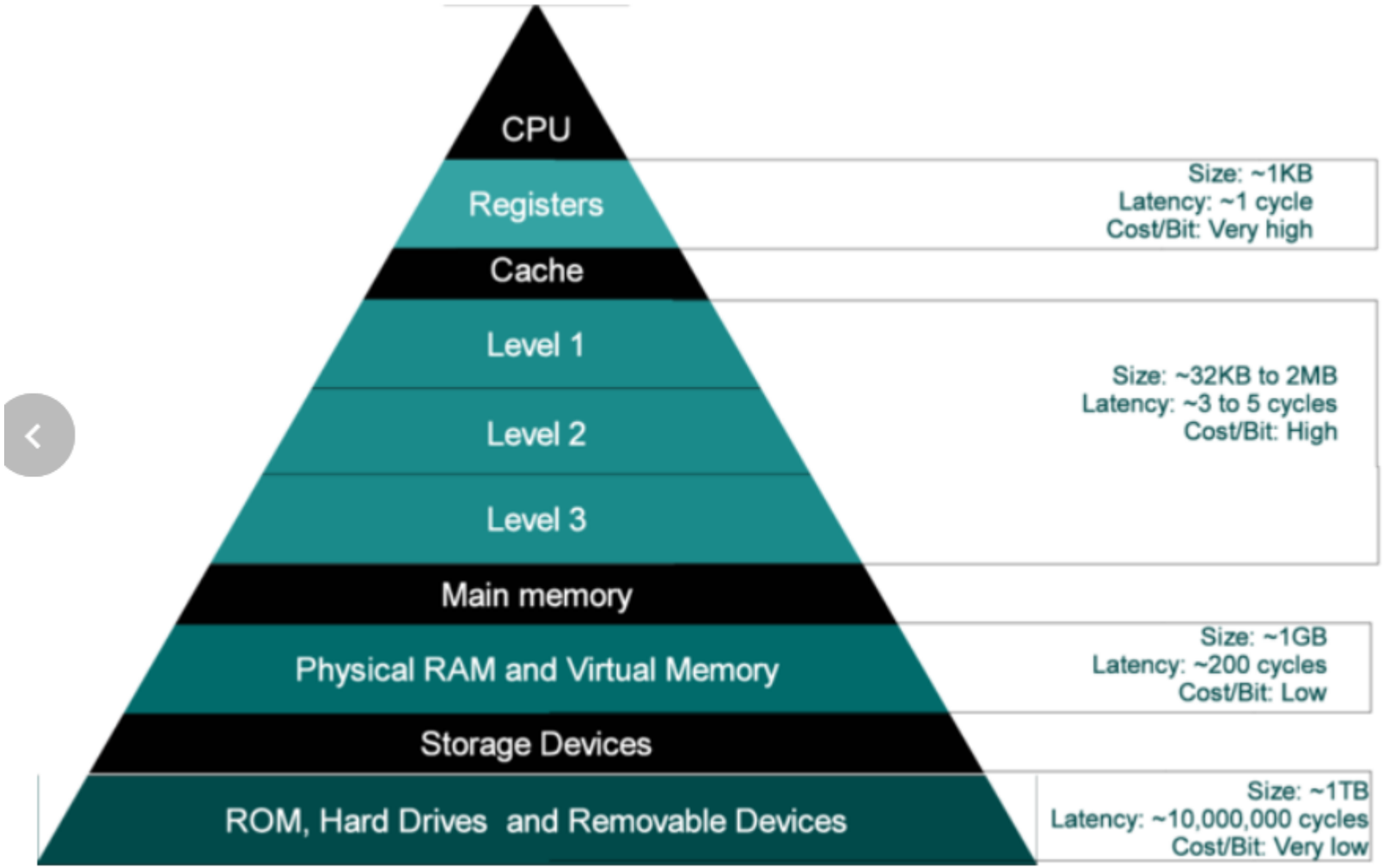
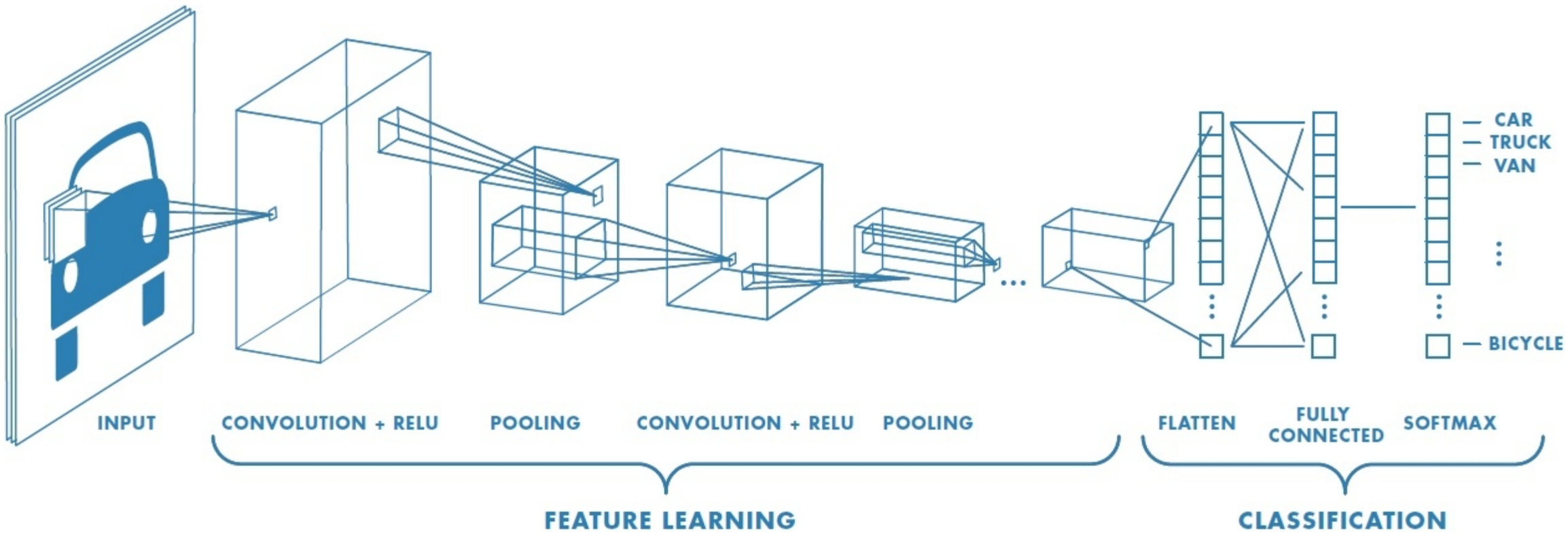


Figure 8: Diagonal strip matrix transposition.

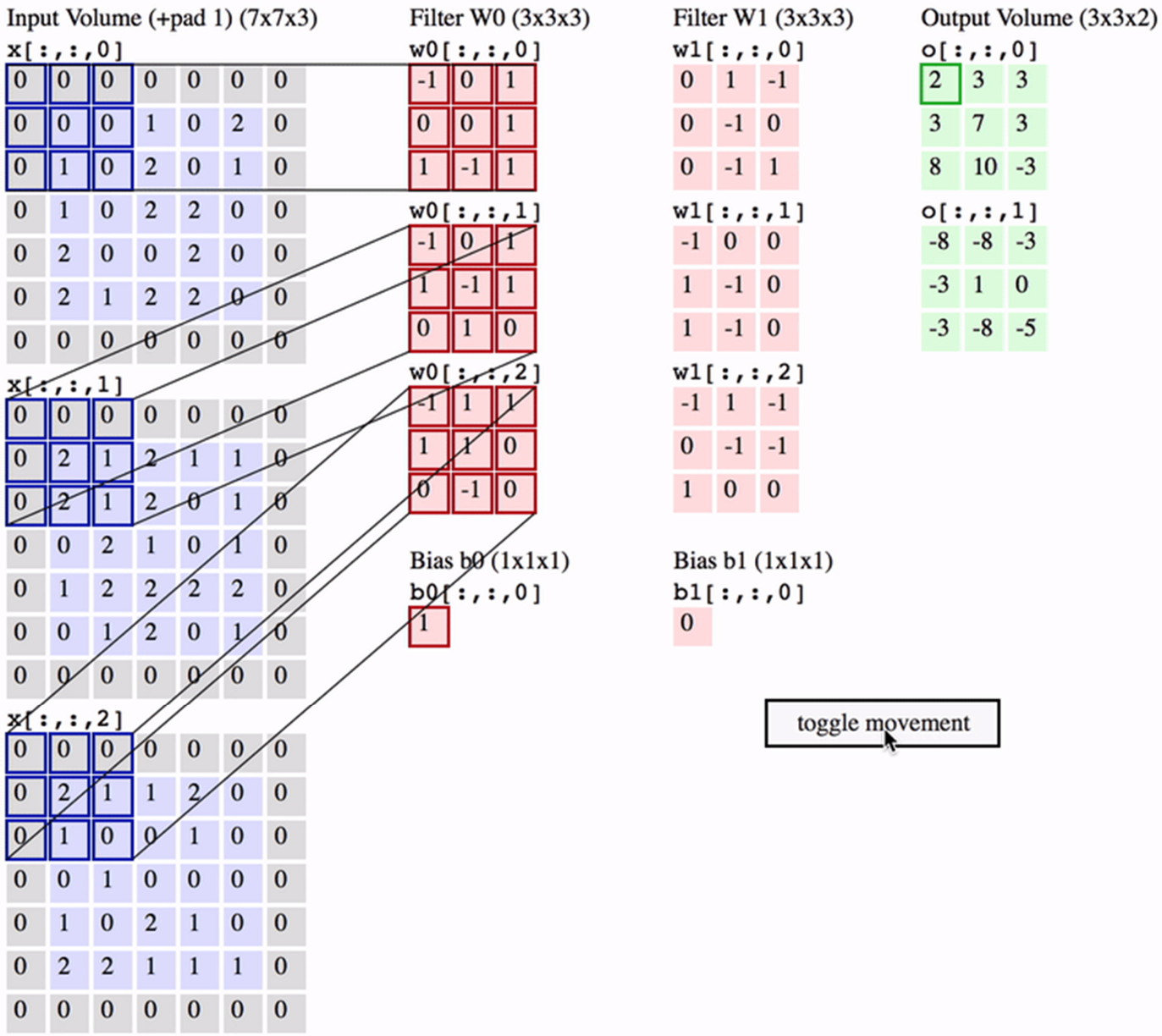
Memory Hierarchy

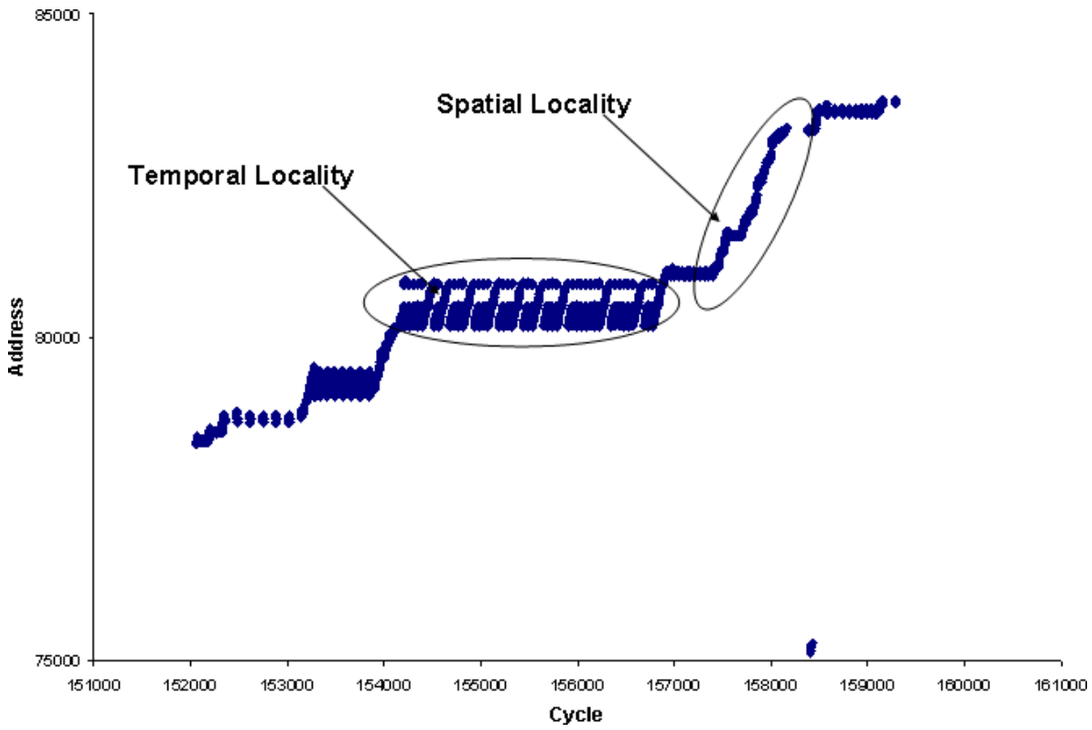


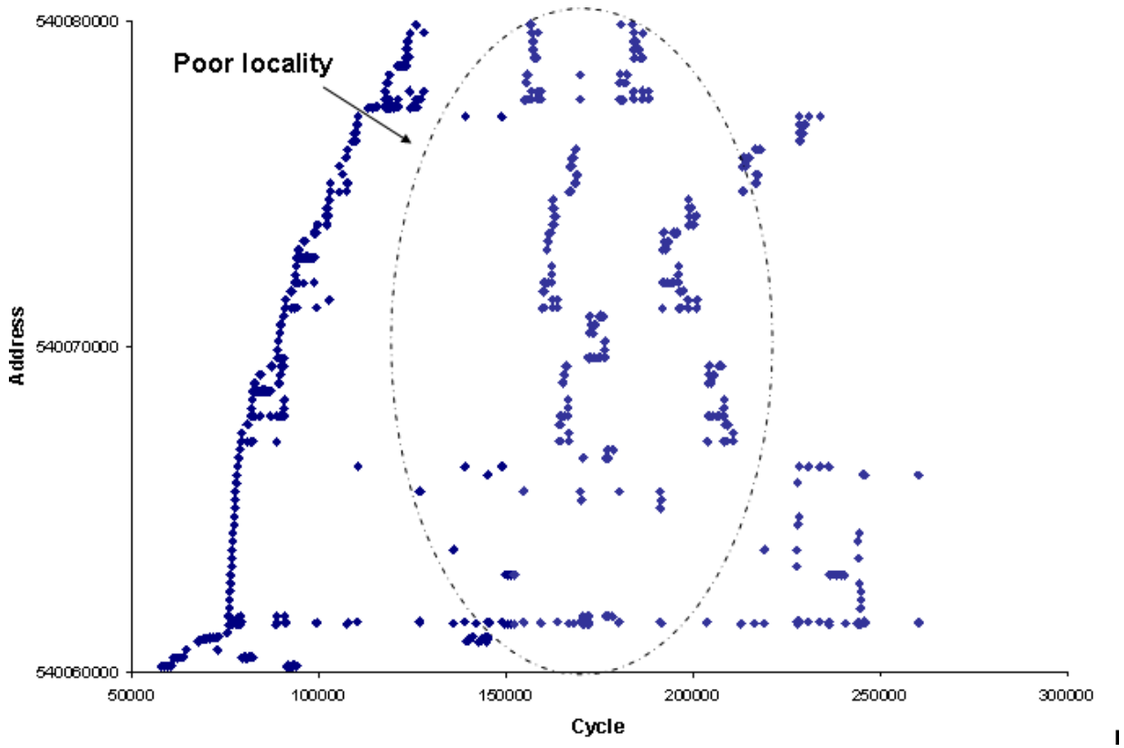
Convolutional Neural Network



https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 0 0 1 0 2 0
0 1 0 2 0 1 0
0 1 0 2 2 0 0
0 2 0 0 2 0 0
0 2 1 2 2 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 2 1 2 1 1 0
0 2 1 2 0 1 0
0 0 2 1 0 1 0
0 1 2 2 2 2 0
0 0 1 2 0 1 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 2 1 1 2 0 0
0 1 0 0 1 0 0
0 0 1 0 0 0 0
0 1 0 2 1 0 0
0 2 2 1 1 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
-1 0 1
0 0 1
1 -1 1
w0[:,:,1]
-1 0 1
1 -1 1
0 1 0
w0[:,:,2]
-1 1 1
1 1 0
0 -1 0
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
0 1 -1
0 -1 0
0 -1 1
w1[:,:,1]
-1 0 0
1 -1 0
1 -1 0
w1[:,:,2]
-1 1 -1
0 -1 -1
1 0 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
2 3 3
3 7 3
8 10 -3
o[:,:,1]
-8 -8 -3
-3 1 0
-3 -8 -5
toggle movement





https://www.embedded.com/print/4017551

OpenCL Memory model