Introduction to EVE/DSP (Features, Partitioning of Algorithms)

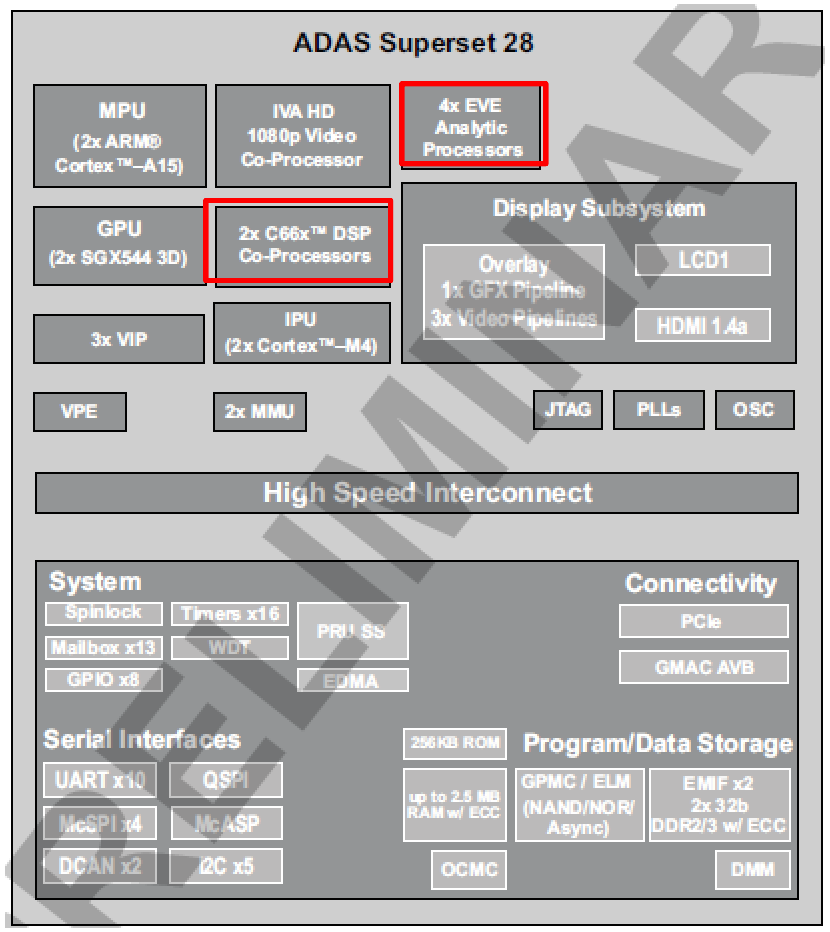
TEXAS INSTRUMENTS

Agenda

- Introduction of VLIW & SIMD
- Introduction of EVE & DSP
- Vision Processing Blocks Mapping
- Use Case Study of TI-PD Algorithm
- Few Concept for Algorithm Development

Introduction

- Lets look at the block diagram of ADAS SOC TDA2x (Vayu)
- It has multiple heterogeneous programmable cores
  - EVE, DSP, A-9, GPU
  - Each of them have further details to be known such as cache, DMA, internal memories, L1, L2…
- To make use of the device optimally, it requires good knowledge and thought process during design stage of the software and certain methodologies and guidelines
- This material tries to introduce high level compute capability of C66xDSP & EVE, and introduces some guideline for partitioning a given algorithm on EVE & DSP.

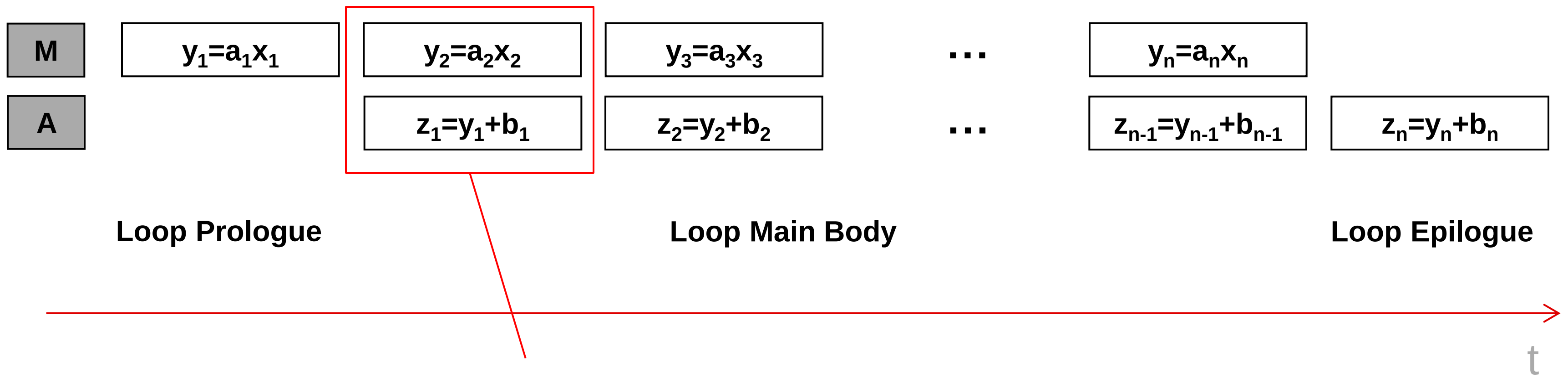


Texas Instruments

Valuable Architecture Elements: VLIW

- VLIW – Very Long Instruction Word
- Pipelined parallel processing

Task: $z_i = a_i x_i + b_i$ $(i = 1,2, \dots , n)$



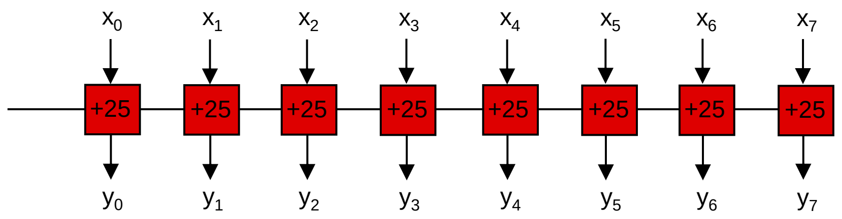
Instructions (parallel processed)

TI-C66x, C67x DSP

Valuable Architecture Elements: SIMD

- SIMD – Single Instruction Multiple Data
- Array processing

Task: y = x + 25 (x, y are arrays)

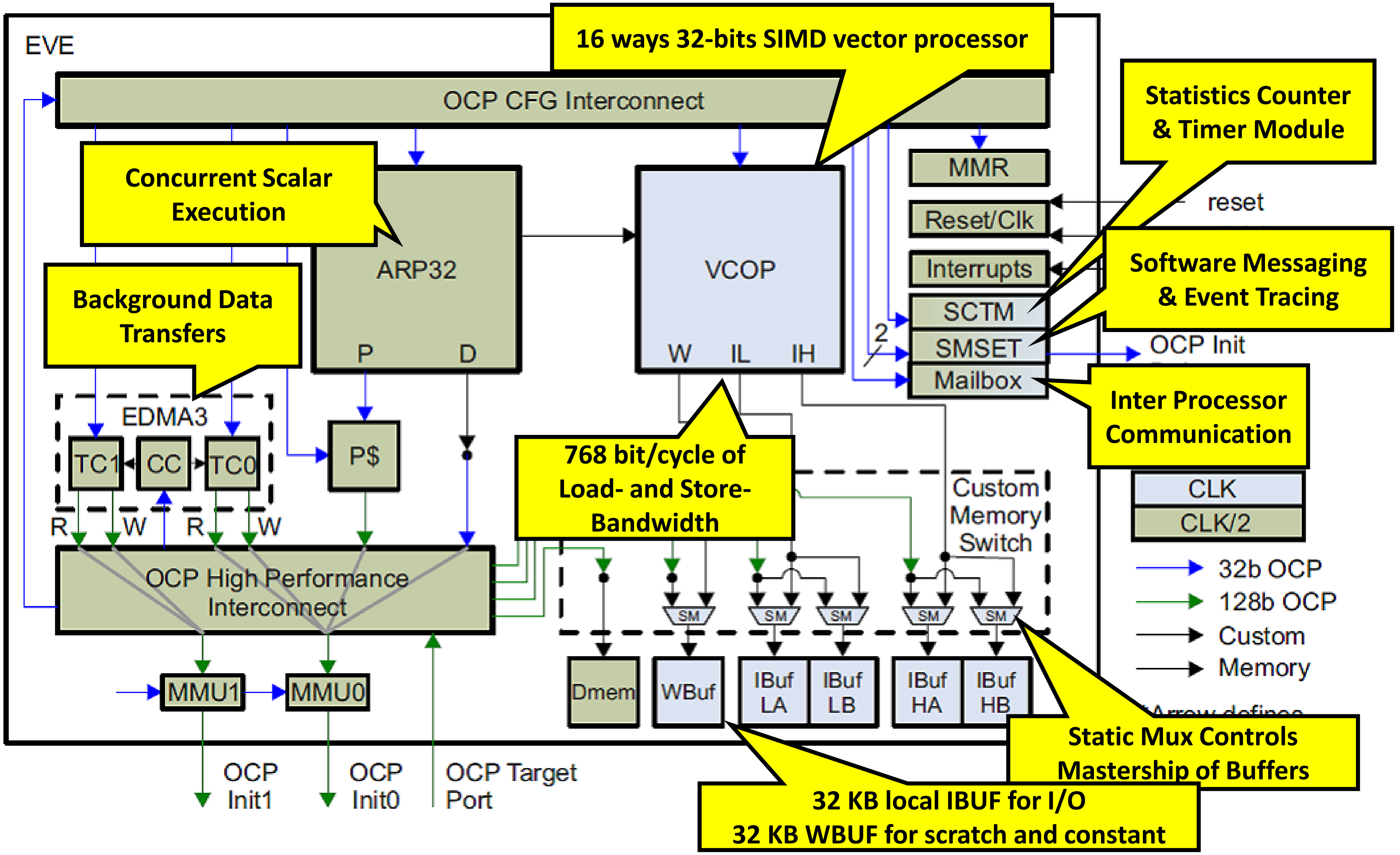


TI- VCOP

TDA2x Compute Processors

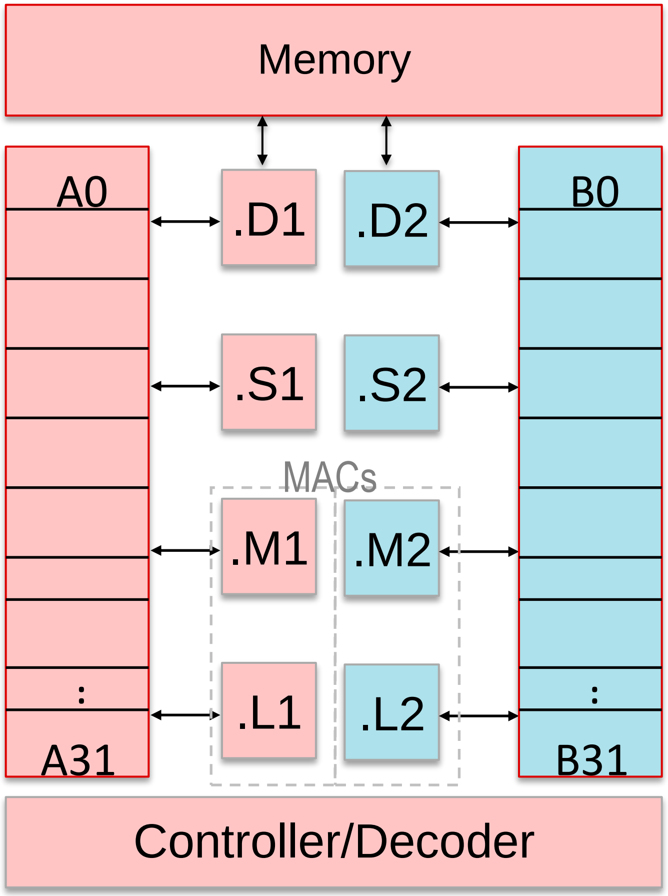
- At a first stage, you need to understand the TDA2x key compute processors
  - EVE
    - Vector Engine (SIMD) with 8-way dual operation issue slot
    - Operations on 8 bit, 16 bit and 32 bits with intermediate results in 40bit precision (multiply operands limited to 16 bit)
    - 16 16x16 bit multipliers, with add/subtract at 500 MHZ within 8 GMAC/sec throughput
    - Fixed point processor (no Floating Point support)
    - Program Cache, No Data Cache
    - Special hardware for Look up Table, Histogram
    - Very efficient for low level (full image based) vision function – filters, feature detectors, Dense processing - feature compute
  - DSP
    - Advanced VLIW CPU with 8 Functional Units
    - 8/16/32/64/128 -bit data support
    - Eight 32-bit / Eight 16-bit / Sixteen 8-bit Multiply per cycle
    - Floating point processor
    - Program Cache, Data Cache
    - Can be used as low, mid and high level vision processing

EVE – Elements and Topology

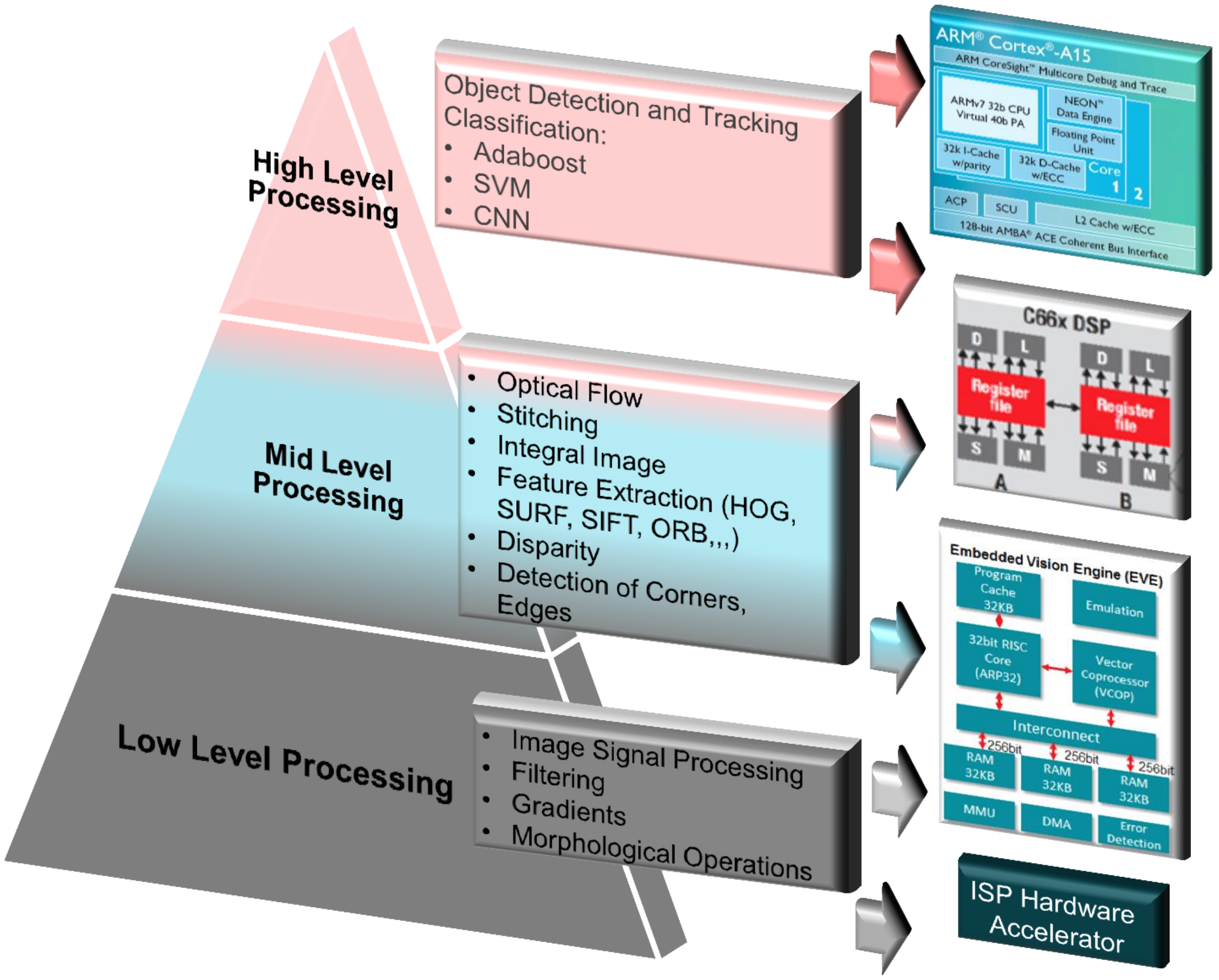

EVE
16 ways 32-bits SIMD vector processor
OCP CFG Interconnect
Statistics Counter & Timer Module
Concurrent Scalar Execution
ARP32
P
D
VCOP
W
IL
IH
MMR
Reset/Clk
reset
Interrupts
SCTM
SMSET
Mailbox
2
OCP Init
Software Messaging & Event Tracing
Inter Processor Communication
Background Data Transfers
EDMA3
TC1
CC
TC0
P$
R
W
R
W
768 bit/cycle of Load- and Store-Bandwidth
Custom Memory Switch
CLK
CLK/2
32b OCP
128b OCP
Custom
Memory
OCP High Performance Interconnect
SM
MMU1
MMU0
Dmem
WBuf
IBuf LA
IBuf LB
IBuf HA
IBuf HB
Static Mux Controls Mastership of Buffers
OCP Init1
OCP Init0
OCP Target Port
32 KB local IBUF for I/O
32 KB WBUF for scratch and constant

DSP Subsystem

- **Available internal Memories**
  - **L1D** : **32 KB** (Configurable as SRAM/CACHE)
  - **L2**: **256 KB**(Configurable as SRAM/CACHE) + 32 KB SRAM
  - **L1P$: 32 KB** Program Cache
- CPU
  - VLIW (Very Large Instruction Word) architecture:
    - Two (almost independent) sides, A and B
    - 8 functional units: M, L, S, D
    - **Up to 8 instructions sustained dispatch rate**
  - Very extensive instruction set:
    - Fixed-point and floating-point instructions
    - More than 300 instructions
    - 8-/16-/32-/64-/128-bit data support
    - Eight 32-bit / Eight 16-bit / Sixteen 8-bit Multiply per cycle
- EDMA with 2 transfer controllers



Vision Processing Mapping



ARM/DSP for Control & High-level vision stages

ARM Cortex Axx:
Scalable RISC
Data Fusion
Memory Coherency

DSP:
VLIW SIMD+MIMD
Data Fusion

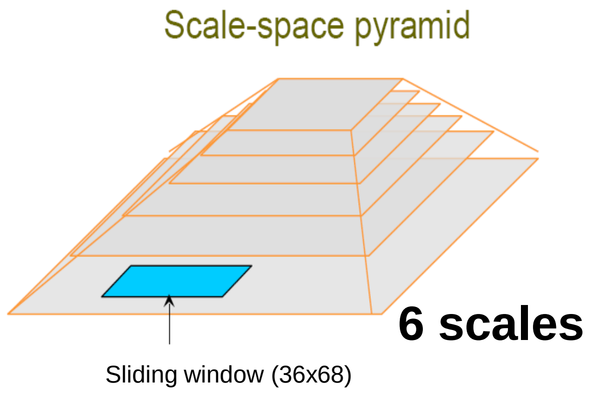
EVE Vector Coprocessor:
High Bandwidth Pixel Operations
SIMD Parallelism
Energy Efficiency

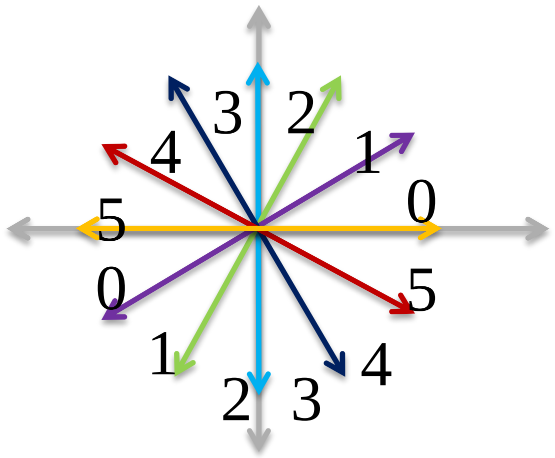
Hardware Acceleration:
High Bandwidth Pixel Operations
Configurable HW Acceleration

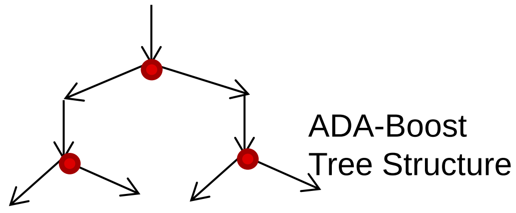
Texas Instruments

TI PD – Case Study for Dev Approach

- Base resolution is assumed to be 1280x720
- The ratio chosen between successive scales is 1.12
- This implies that it takes 6 scales to reach the next octave
- Pedestrian model (Sliding window) used 36x68
- Sliding window step size used 4x4
- Cell overlap is 4x4
- Re-size will be done till width >= 36 and height >= 68 (total 23 scales)
- The algorithm uses **<u>10 feature planes</u>**
  - Y,U,V → Summation over 8x8
  - Gradient magnitude → Summation over 8x8
  - HOG for 6 bins between 0 -180 degree
  - – Cell size = 8x8
  - 8*16*10→ 1280 features per position
  - 1 position every 4x4 block in each scale
- Uses Adaboost classifier with 2 Level trees

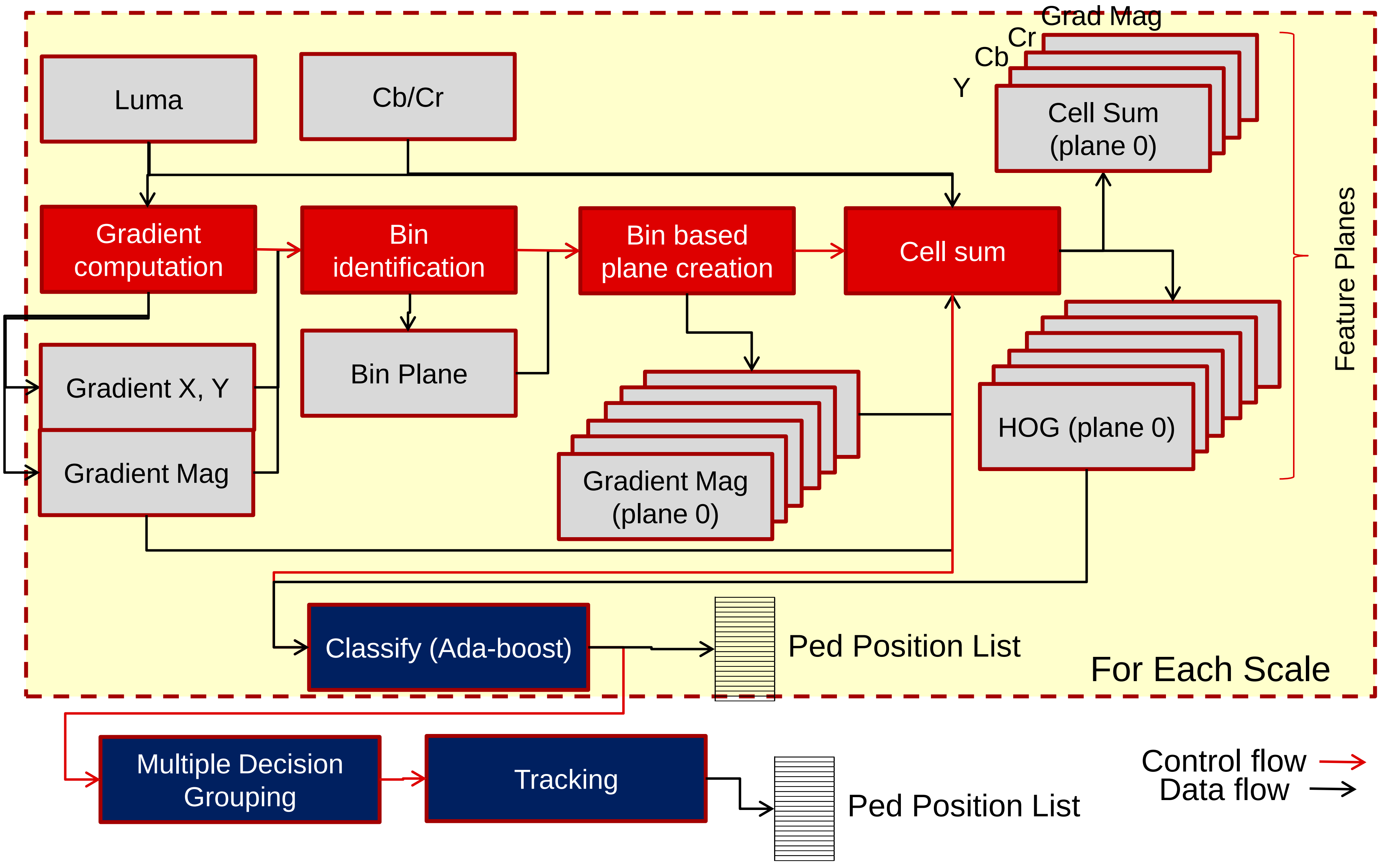






Refer below Links for more details on HOG and Pedestrian detection algorithms
http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf
http://iica.de/pd/slides/hog.ppt

Pedestrian Detection - Kernels



Processing blocks Properties

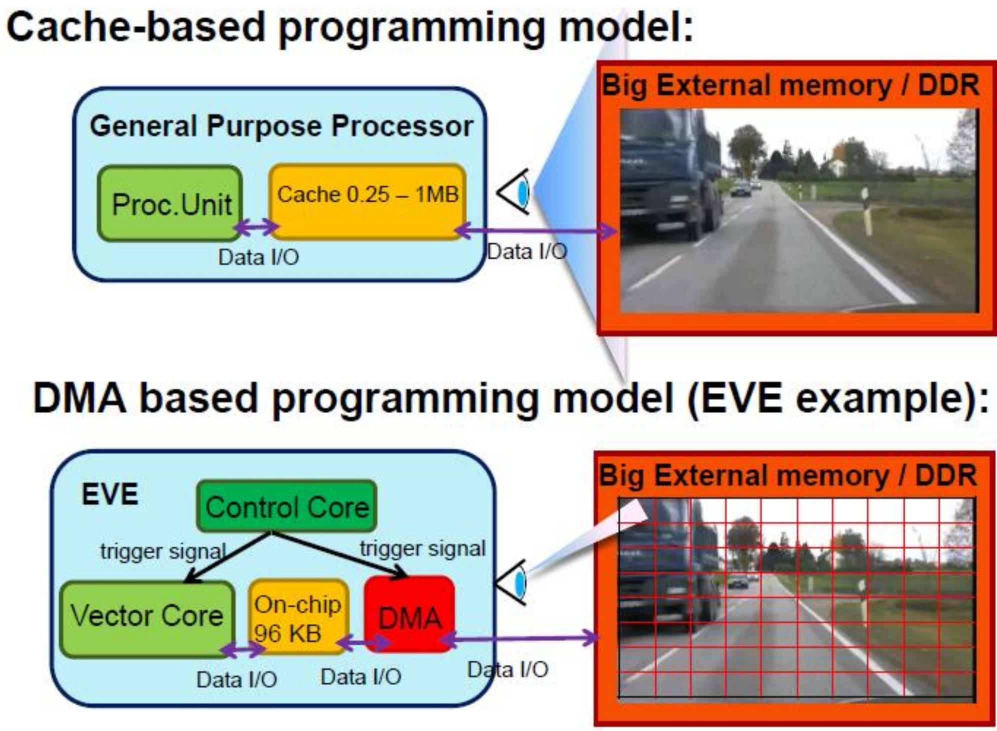
- Gradient computation, Bin identification, plane creation, Cell sum
  - Pixel based dense processing
  - No Floating point usage
- Scale Creation (Resizing)
  - Pixel based dense processing
  - No Floating point usage
- Ad boost
  - Pixel based dense processing
    - Soft cascade - early exit makes it non-dense, few points are processing more trees and few are less
  - No Floating point usage
- Window Grouping, Tracking
  - Control Code
  - Floating point

Processing blocks Properties -- Mapping

• Gradient computation, Bin identification, plane creation, Cell sum – Pixel based dense processing – No Floating point usage	EVE
• Scale Creation (Resizing) – Pixel based dense processing – No Floating point usage	EVE / VPE
• Ad boost – Pixel based dense processing • Soft cascade - early exit makes it non-dense, few points are processing more trees and few are less – No Floating point usage • Window Grouping, Tracking – Control Code – Floating point	DSP

Data Access Approach

- Cache Approach (Frame based)
  - Processing unit "sees" the entire image in DDR through cache.
  - Programming model is easier but inefficient in term of power and performance, because 2-D access patterns generates frequent cache misses.
- DMA Approach (Block based)
  - Core processes data from on-chip memory
  - DMA used to read/write small blocks of image in DDR from/to on-chip
  - Control core coordinates DMA and processor for maximum parallelism
  - DMA approach is often preferred for low level image processing
  - In case of EVE, it is the only choice in absence of data cache and VCOP's ability to see DDR .


Cache-based programming model:
General Purpose Processor
Proc.Unit
Cache 0.25 – 1MB
Data I/O
Data I/O
Big External memory / DDR
DMA based programming model (EVE example):
EVE
Control Core
trigger signal
trigger signal
Vector Core
On-chip 96 KB
DMA
Data I/O
Data I/O
Data I/O
Big External memory / DDR

FEW CONCETPS

- HOST Emulation
- Data Flow Design Decision

Host Emulation

- EVE and C66x both have a C compiler, one can write a C Code and directly port to these targets
- But utilization of processor capabilities might not be best as it demands specific optimizations
- C66x optimization involves
  - Form the loops for key processing blocks
  - Usage of specific instructions to accelerate the processing
- EVE optimization involves
  - Form the loops for key processing blocks
  - Use of "VCOP Kernel C (subset of C++)" language for loops
- In both cases the second step makes the code non-familiar for any other C compiler to use during PC development
- Facilitated with **Host Emulation (Instruction Set emulation)**
  - In DSP – Intrinsic can be implemented as function calls, host emulation package (details)
  - In EVE – "VCOP Kernel C" is facilitated by compiler by providing "vcop.h" to emulate "VCOP Kernel C" as a C++ program on PC
- DMA
  - DMA software functions can be created which emulates the hardware behavior, we have created such functions use them during software development

So entire algorithm can be executed on PC

Data Flow Design Decisions

- Lets say Function2 is successor of Function1
- While connecting them, we have 2 choices
  A. Operate Function1 on complete frame – send output data to DDR* – Call Function2 with the data in DDR
  B. Call function 1 on small block in the frame – keep data in internal memory – Call function 2 with the data in internal memory
- Efficiency (power, performance) point of view choice B is always best
- But it also depends on the algorithm property
  - Possible to break?
  - Data Dependency from successor to predecessor?

*Assuming that frame data is huge and can not fit in device internal memory

Possible to break

- A function is possible to operate on smaller data blocks if there is no global dependency. In this case, it can be called multiple times with smaller data blocks to complete entire frame processing
    - Example : sorting (sorting is not a candidate to break into smaller parts)
    - Example: Gradient of a frame can be computed by computing gradient on smaller blocks
- Some times even the algorithm is possible to break but the data overlap is very high in successive blocks, which makes lot of data re-fetch and re-compute (example filters with very high number of taps)
    - Under such situations, it is also good to consider not to operate on small blocks

Dependency

- Some times Function2 and Function1 might be good candidate to break, even though algorithm property might not allow them to operate at smaller data granularity
- These are the situations where Function2 depends on Function1 processing on complete frame?
  - Example
    - Contrast enhancement:
      - Compute histogram and find min, max intensity (Function 1)
      - Map min to 0, max to 255 and all intermediate values appropriately (Function 2)
    - In this case Function2 can only start after Function1 does the processing on complete frame
  - Example
    - Gradient computation → Bin identification → plane creation → Cell sum
    - There is no global dependency from previous function – so can be connected at smaller blocks retaining the intermediate outputs in internal memory

Easy to Integrate

- **Resource Sharing**
  - Algorithm use system resources, these are shared with other component in system
    - Processor
    - Memory
    - DMA
  - So it is important to have a centralized resource manager in system outside the algorithm (part of system software) to manage these resources
  - Algorithm should have mechanisms to request its resource needs to system software
- Abstracted Standardized interface
  - Standard set of API across multiple set of algorithm helps system integrator to save time in understanding the interface of an algorithm
  - Consists of set of rules & guidelines to ensure seamless integration and resource sharing of several algorithms with minimal/no overhead
  - Standard handshake mechanism for resource management (int. memory, DMA) across algorithm instances, controlled by centralized application (not by algo)

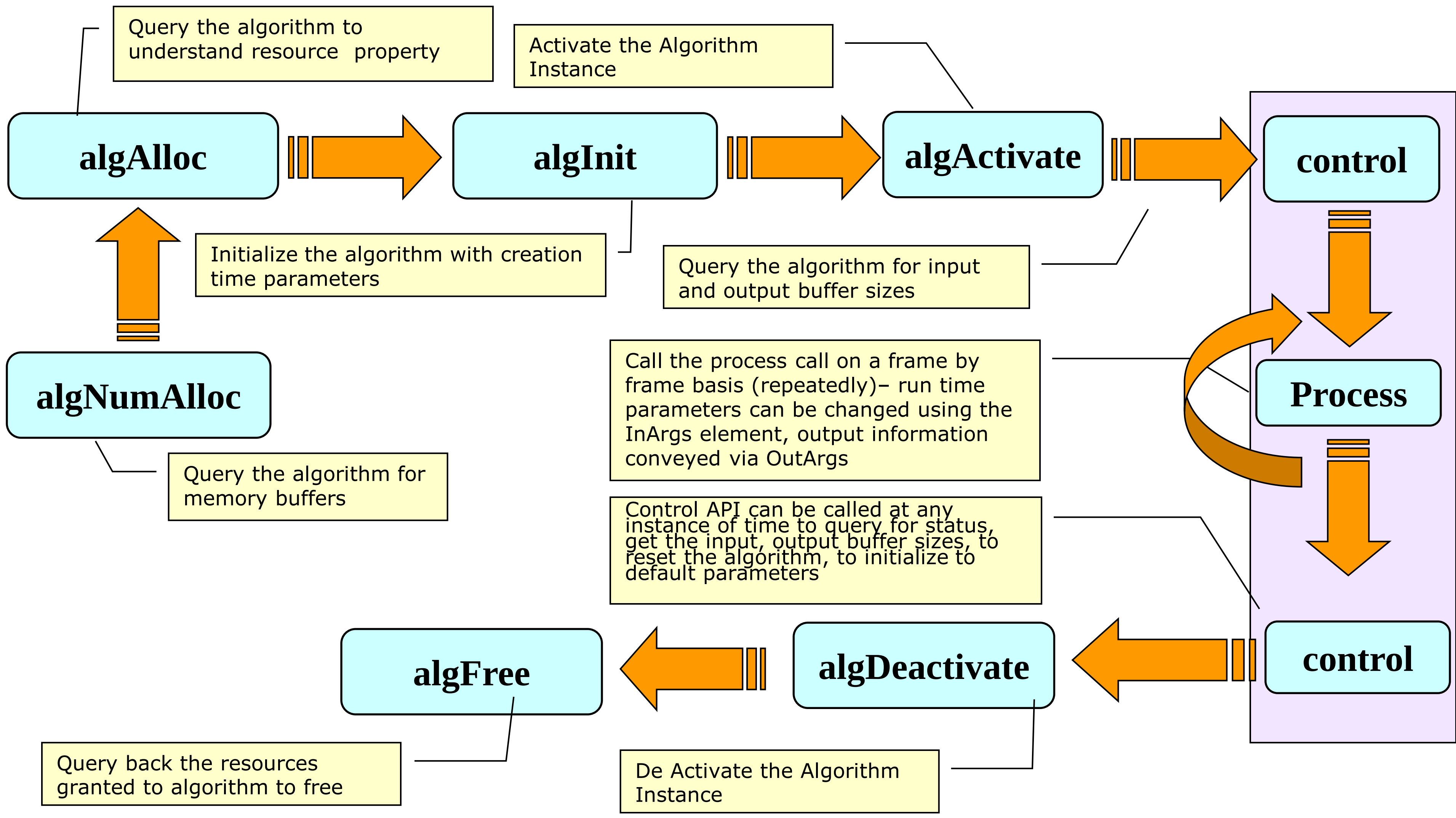
Memory (Resource)

- Memory can be internal, external
  - Internal memories are mostly used as scratch, if algorithm is using it as persistent – should provide some functions to store/re-store so other algorithm in system can re-use it
- External memories used in algorithms can be categorized in 2 parts
  - Input and Output
    - Input and Output are known by the system and algorithm just consume it
  - Internal memories required by algorithm for its private data
    - Internal memory property are not known by system and should be explicitly requested by algorithm to system

DMA (Resource)

- DMA is also shared resource for multiple algorithms on the sub system
- It can also be managed by system same as memory
- TI devices have local EDMA for most of its compute sub-system and other system component might not require to use these subsystem DMAs at system level DMA also exist
- Under this situation, a solution might decide that sub system DMAs are owned by algorithm executing on that subsystem and can be assumed to be solely available
  - So for this kind of design algorithm might not request DMA to system
  - But such design decisions should be aligned at system level before making this assumption

Abstracted Standardized interface (xDAIS)


Query the algorithm to understand resource property
Activate the Algorithm Instance
algAlloc
algInit
algActivate
control
Initialize the algorithm with creation time parameters
Query the algorithm for input and output buffer sizes
algNumAlloc
Call the process call on a frame by frame basis (repeatedly)– run time parameters can be changed using the InArgs element, output information conveyed via OutArgs
Process
Query the algorithm for memory buffers
Control API can be called at any instance of time to query for status, get the input, output buffer sizes, to reset the algorithm, to initialize to default parameters
algFree
algDeactivate
control
Query back the resources granted to algorithm to free
De Activate the Algorithm Instance

Development Guidelines

- Read TI's tools, processor and training material on DSP, EVE and SOC
- Get the first C code ported – use DSP or A15
- Have performance measurement of the different critical blocks. Record it in a table
- Prepare an algorithm document – it should mention about key processing blocks (operations per pixel, dependency of information) and operating data width(8-bit, 16-bit, 32-bit) of these blocks clearly
- Based upon understand of EVE and DSP put a design (data flow and core partitioning) with performance estimates (use resources such as VLIB Data sheet, EVE SW Data sheet)
  - Start with DSP– to keep things simple in case EVE learning curve is higher
- Setup a meeting with TI experts – present algorithm details, get the design reviewed and seek feedback such as
  - What is already available and can be reused
  - Design changes for optimal SOC use
  - Comments on performance estimates {Align on timeline for this meeting 2-3 weeks ahead}.
  - Meeting should be led by Key responsible technical person from 3rd party for this algorithm and that person should have good working experience of signal processing algorithm on embedded processors.
- Based upon the feedback – redesign and put a plan of activities
- Meet once again to get the plan and new design reviewed
- Seek help via E2E/FAE as appropriate

References

- Specifications
  - TMS320C6000 Programmer’s Guide (spru198k)
  - http://processors.wiki.ti.com/index.php/Run_Intrinsics_Code_Anywhere
  - ADAS Superset 28 Technical Reference Manual(SPRUHK5G)
  - EVE Programmer’s Guide (SPRUHC1B)
- Software
  - Vision SDK Release (02.05.00.00 onwards)
  - EVE SW Release (01.07.00.00 onwards)
  - DSP VLIB (3.1.0 onwards)