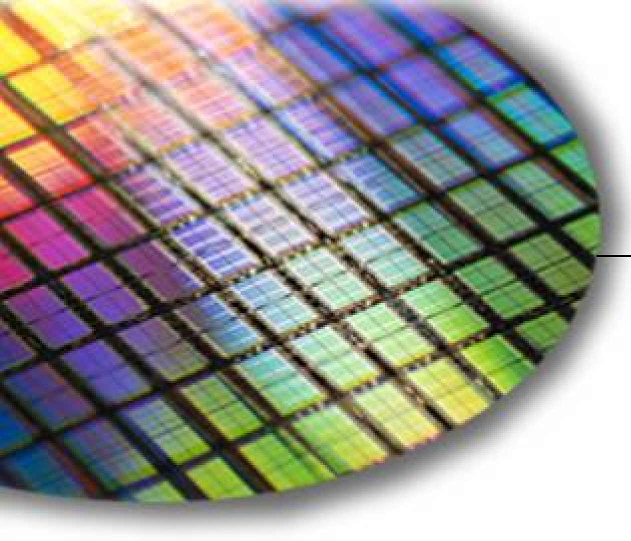


The World Leader in High Performance Signal Processing Solutions

Multi-core programming frameworks for embedded systems

Kaushal Sanghai and Rick Gentile
Analog Devices Inc.,
Norwood, MA

Outline

- Multi-core programming challenge
- Framework requirements
- Framework Methodology
- Multimedia data-flow analysis
- BF561 dual-core architecture analysis
- Framework models
- Combining Frameworks
- Results
- Conclusion

Multi-core Programming Challenge

- To meet the growing processing demands placed by embedded applications, multi-core architectures have emerged as a promising solution
- Embedded developers strive to take advantage of extra core(s) without a corresponding increase in programming complexity
- Ideally, the performance increase should approach "N" times where "N" is the number of cores
- Managing shared-memory and inter-core communications makes the difference!
- Developing a framework to manage code and data will help to speed development time and ensure optimal performance
- We target some compute intensive and high bandwidth applications on an embedded dual-core processor

ANALOG DEVICES

Framework requirements

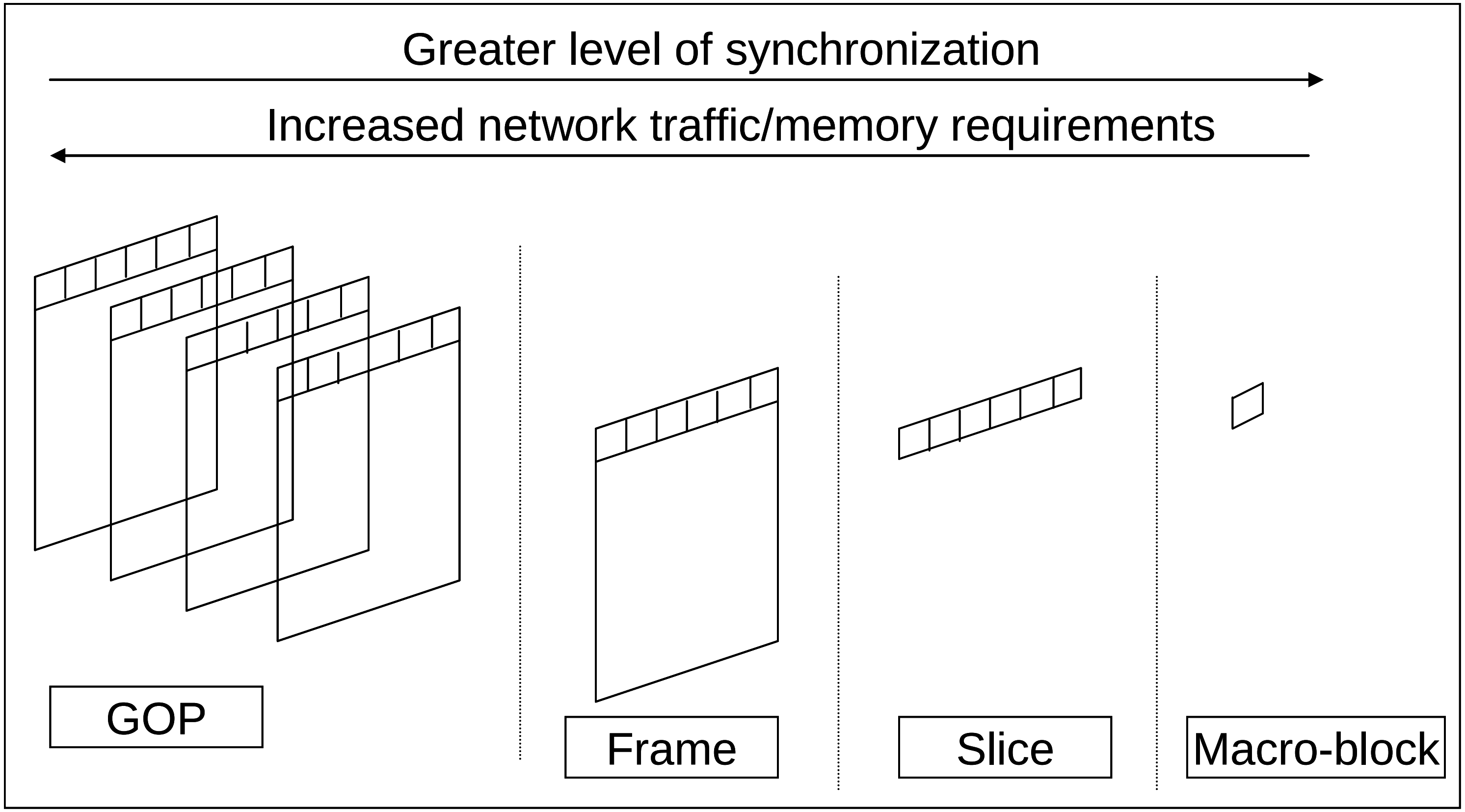
- **Scalable across multiple cores**
- **Equal load balancing between all cores**
- **A core data item request is always met at the L1 memory level**
- **Minimum possible data memory footprint**

Framework methodology

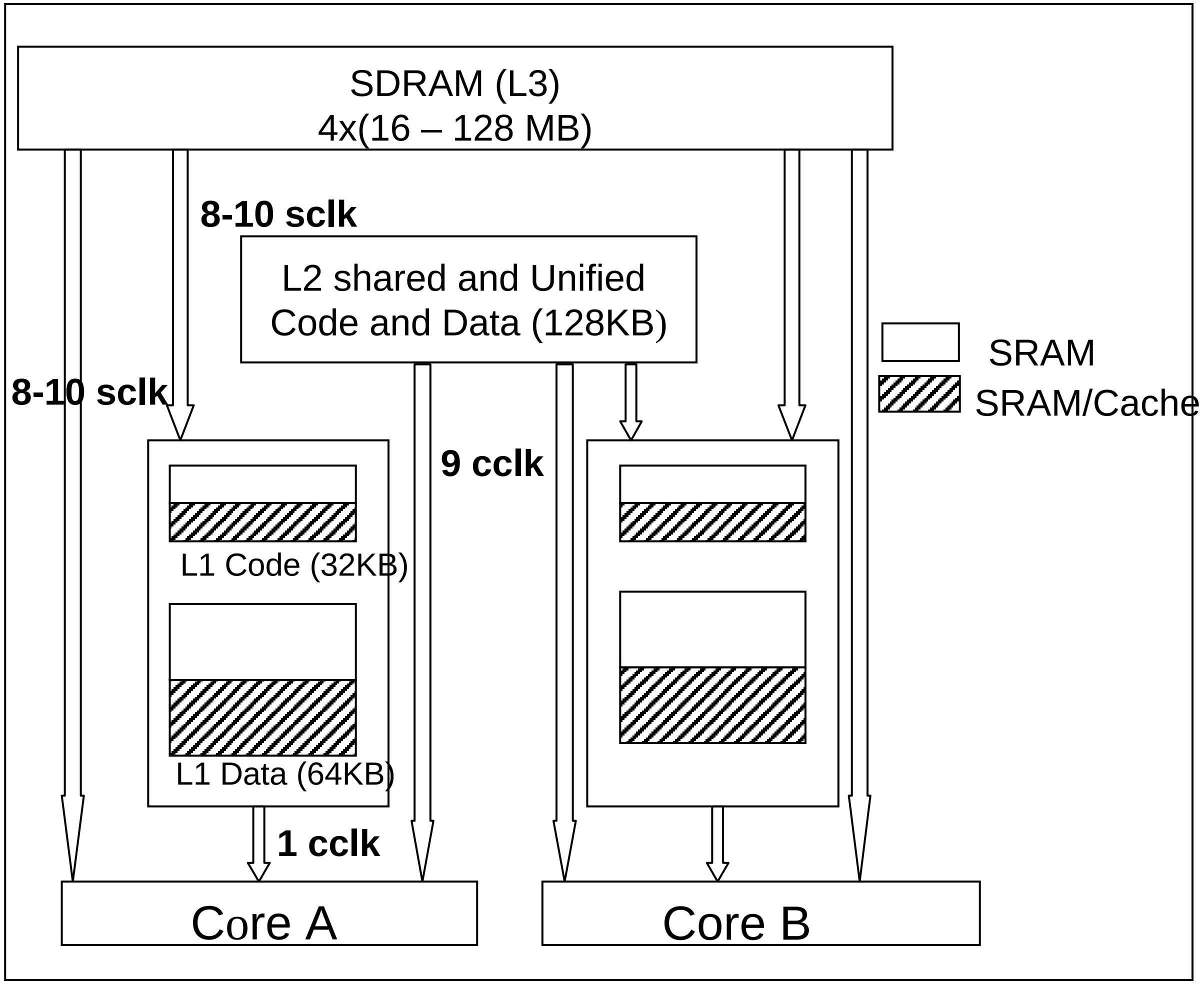
- **Understanding the parallel data-flow of the application with respect to spatial and temporal locality**

- **Efficiently mapping the data-flow to the private and shared resources of the architecture**

Multimedia Data-flow Analysis



ADSP-BF561 Dual-core Architecture Analysis



- **Dual-Core architecture**
- **Private L1 code and Data memory**
- **Shared L2 and external memory**
- **4 Memory DMA channels**
- **Shared peripheral interface**

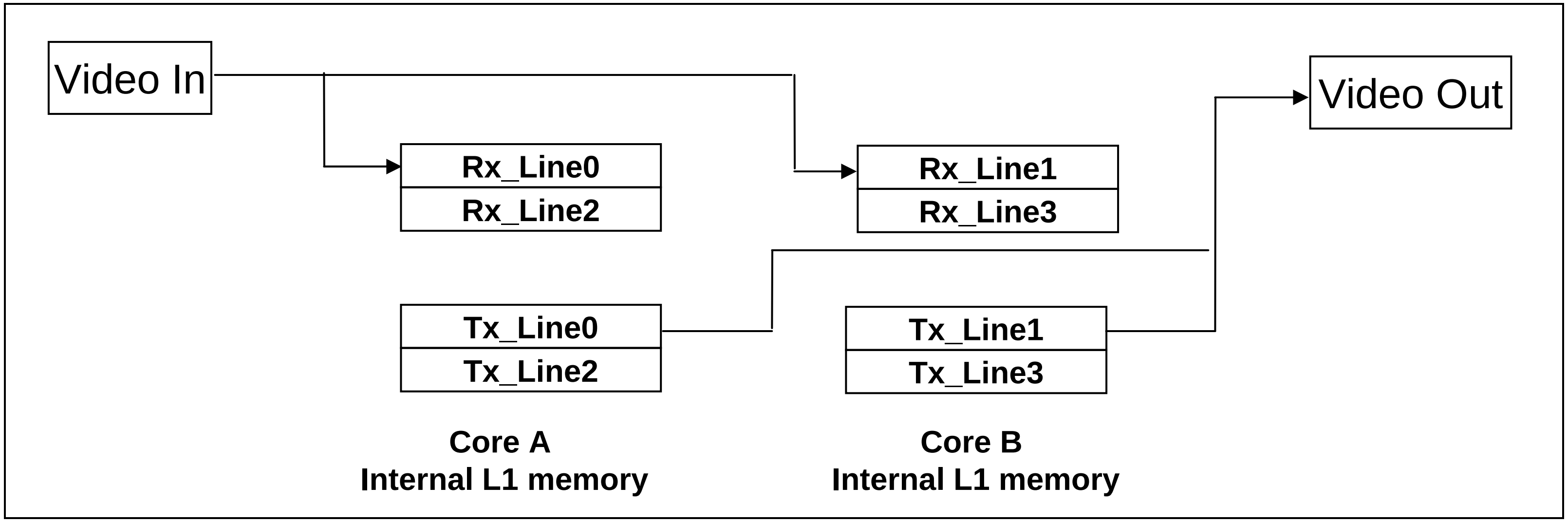
Framework models

- Slice/Line processing
- Macro-block processing
- Frame processing
- GOP processing

Framework design

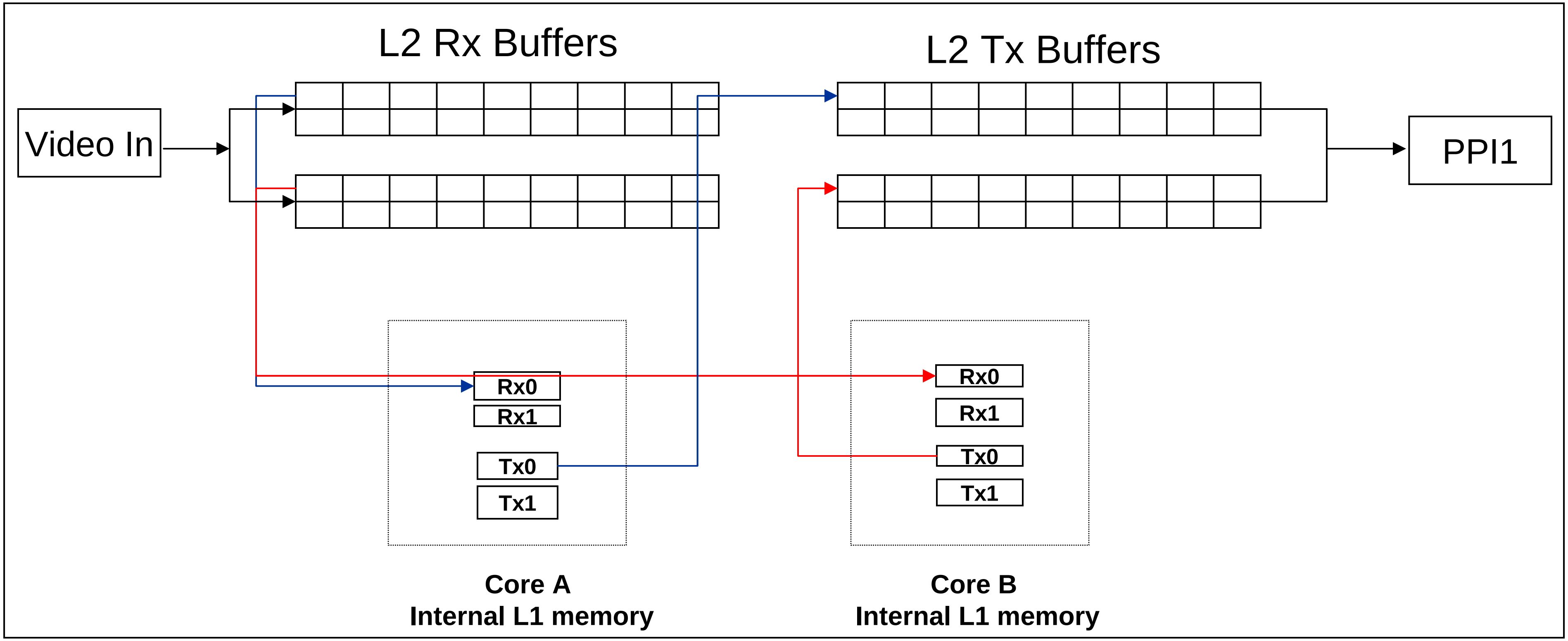
- Data moved directly from the peripheral DMA to the lowest (Level 1 or Level 2) possible memory level based on the data access granularity
- DMA is used for all data management across memory levels, saving essential core cycles in managing data
- Multiple Data buffers are used to avoid core and DMA contention
- Semaphores are used for inter-core communication

Line processing framework



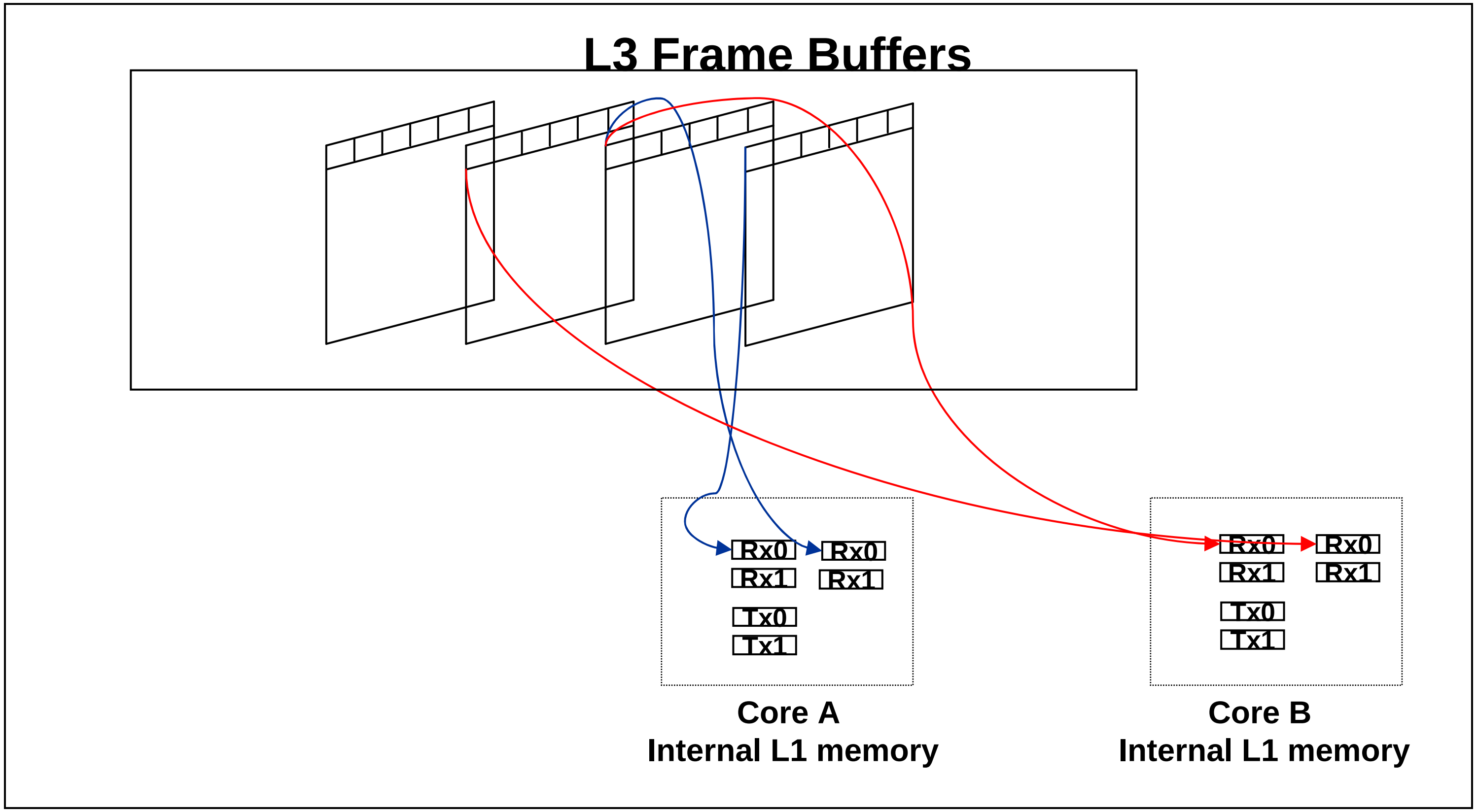
- **No L2 or L3 accesses made, thereby saving external memory bandwidth and DMA resources**
- **Only DMA channels used to manage data**
- **Applicable examples - color conversion, histogram equalization, filtering, sampling etc.**

Macro-block processing framework



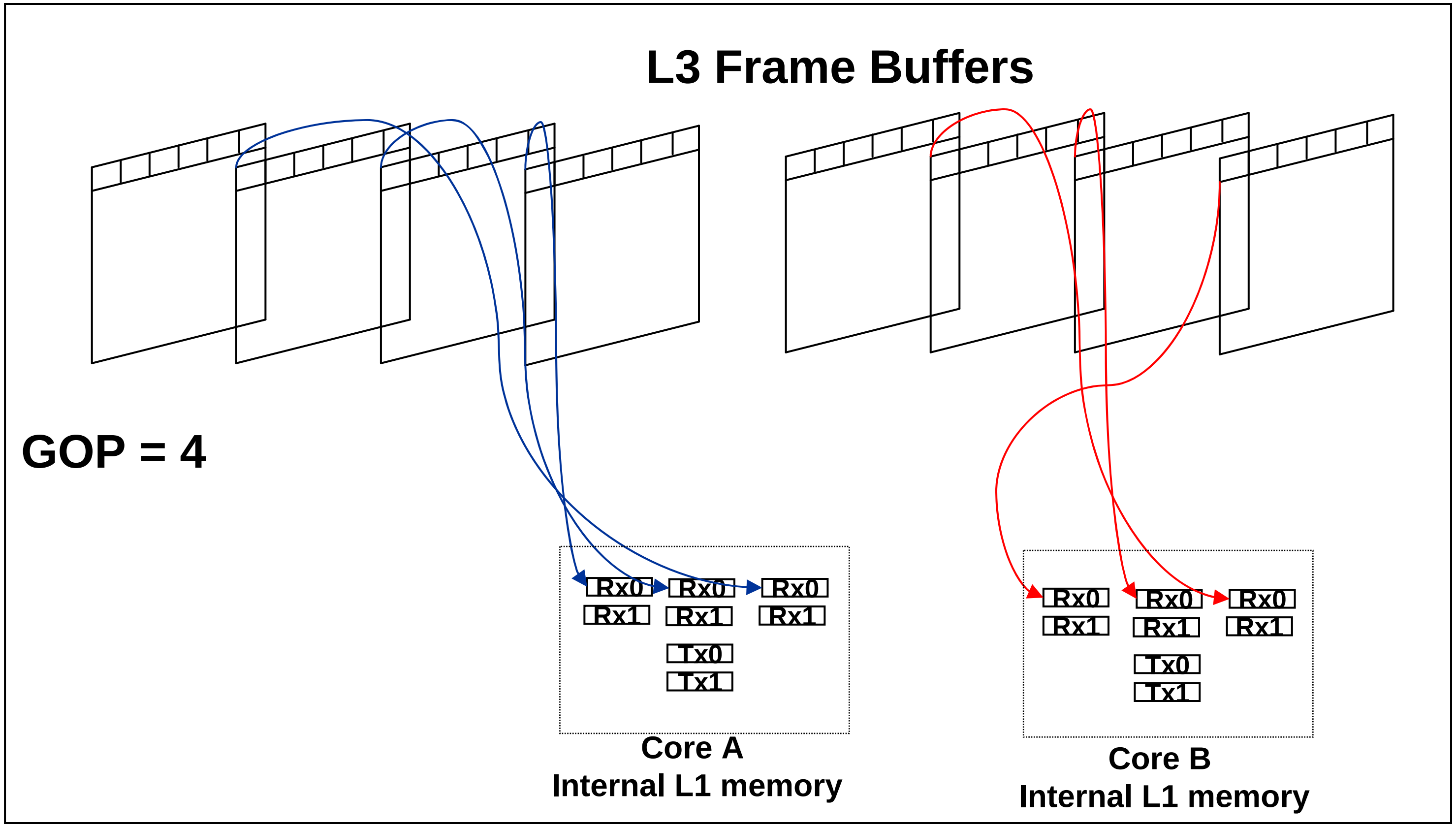
- **No L3 accesses**
- **Applicable examples - edge detection, JPEG/MJPEG encoding/decoding algorithms, convolution encoding etc**

Frame processing framework



- Applicable example - motion detection

GOP processing framework



- **Applicable examples - encoding/decoding algorithms such as MPEG-2/MPEG-4**

Results

Template	Core cycles/pixel*(approx.) single core	Core cycles/pixel*(approx.) - two cores	L1 data memory required(bytes)	L2 data memory required (bytes)	Comments
Line Processing	42	80	(*line size*)*2; for ITU-656 - 1716*2	——	double buffering in L1
Macro-block Processing	36	72	(*Macro-block size(nxm)*)*2	Slice of a frame; (*macro-block height *line size*)*4	double buffering in L1 and L2
Frame processing	35	70	(*size of sub-processing block)*(number of dependent blocks*)	(*size of sub-processing block)*(number of dependent blocks*)	Only L1 or L2 cannot be used double buffering in L1 or L2

Using the Templates

Identify the following items for an application

- The granularity of the sub-processing block in the image processing algorithm
- The available L1 and L2 data memory, as required by the specific templates.
- The estimate of the computation cycles required per sub-processing block
- The spatial and temporal dependencies between the sub-processing blocks. If dependencies exist, then the templates needs modification to account for data dependencies

Conclusion

- **Understanding the data access pattern of an application is key to efficient programming model for embedded systems**
- **The frameworks combine techniques to efficiently manage the shared resources and exploit the known data access pattern in multimedia applications to achieve a 2X speed-up**
- **The memory footprint is equal to the smallest data access granularity of the application**
- **The frameworks can be combined to integrate multiple algorithms with different data access pattern within an application**