Asymmetric MultiProcessing for embedded vision

D. Berardi, M. Brian, M. Melletti, A. Paccoia,
M. Rodolfi, C. Salati, M. Sartori

T3LAB

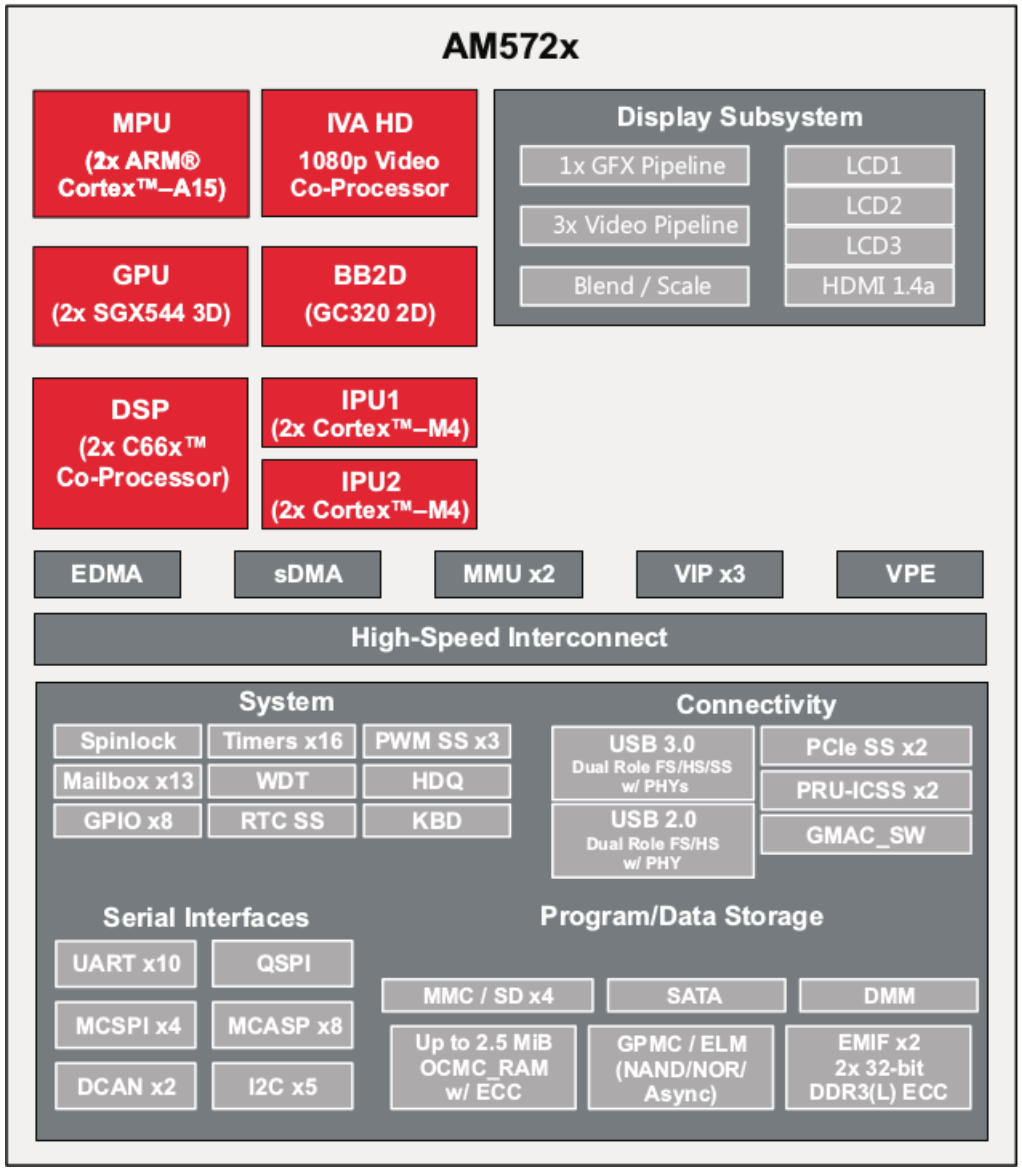
OPEN-NEXT

Bologna, October 18, 2017

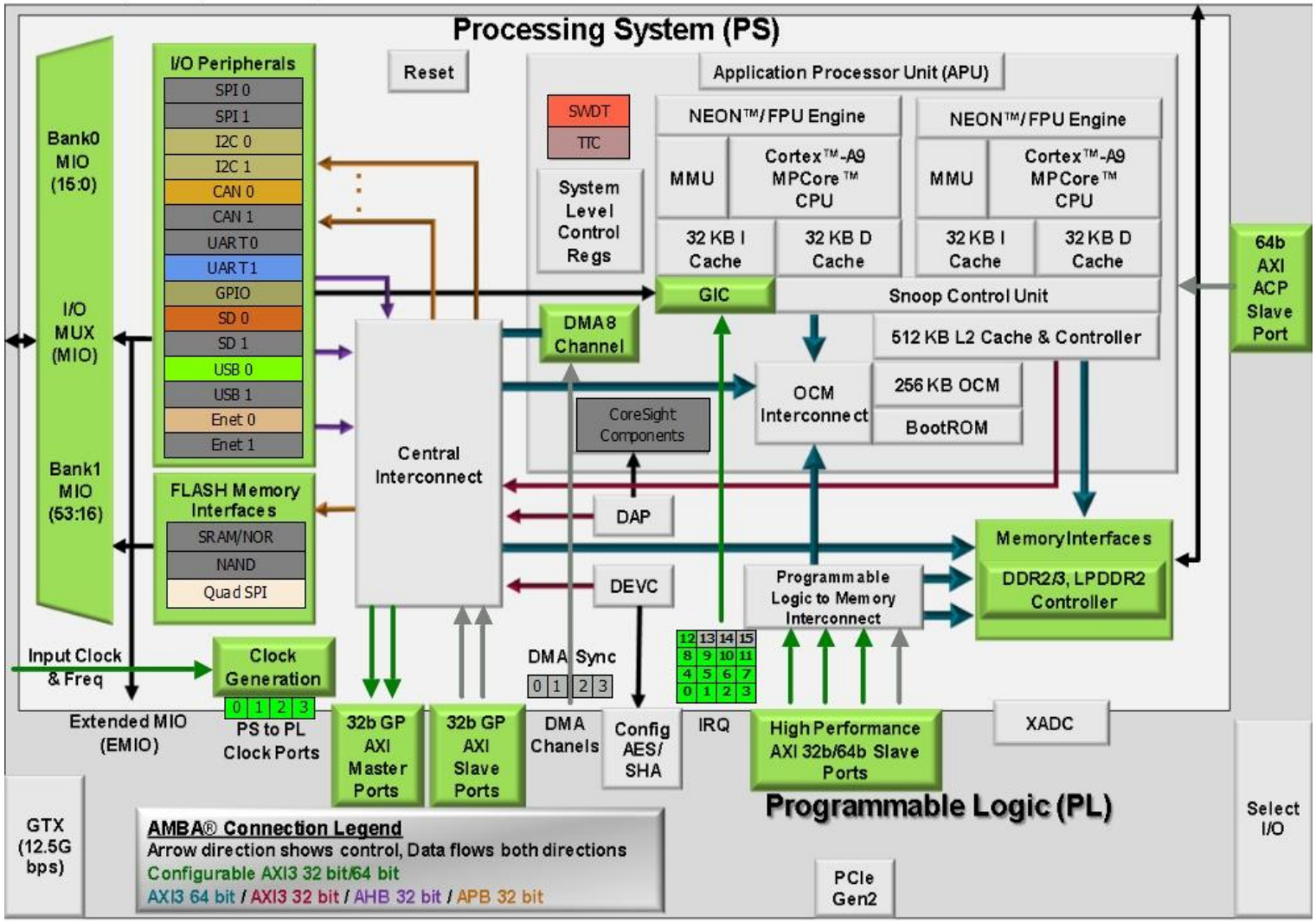
t3lab
TECHNOLOGY
TRANSFER
TEAM

A Linux centered SW infrastructure for the support of asymmetric multi-processing

HW platforms: TI Sitara AM572x



HW platforms: Xilinx Zynq and SW generated heterogeneity



Heterogeneous requirements: computing vs. controlling

- **Computing**
  - Large and complex applications
  - Heavy computational requirements
  - Complex arithmetic
  - Large data movements
  - Real time / high throughput
- **Controlling**
  - Real time / determinism
  - Minimum latency

Complex SW platforms

- Multiple kernels and multiple independent instances of a kernel on the same chip
  - Linux
  - RTOS (e.g. FreeRTOS is not multicore!)
  - Bare metal
- Partitioning of resources
- Boot & life cycle of processing cores and kernels
- Interprocessor/intercore communications
- Interprocess communications (IPC)
- Programming model

AMP support in Linux

- **remoteproc**
  - Life cycle: Coordinated start/stop of different kernel instances
  - Master-slave architecture, Linux can only be master
- **rpmsg**
  - Support of interprocessor/intercore communications
  - Analogous to a *data link service in computer networks*
  - Implemented as a bus sub-system
- Links between remoteproc and rpmsg
  - Allocation and initialization of shared memory communication resources (used by rpmsg) by remoteproc
  - 2 cores can communicate via rpmsg only if one is the remoteproc master of the other

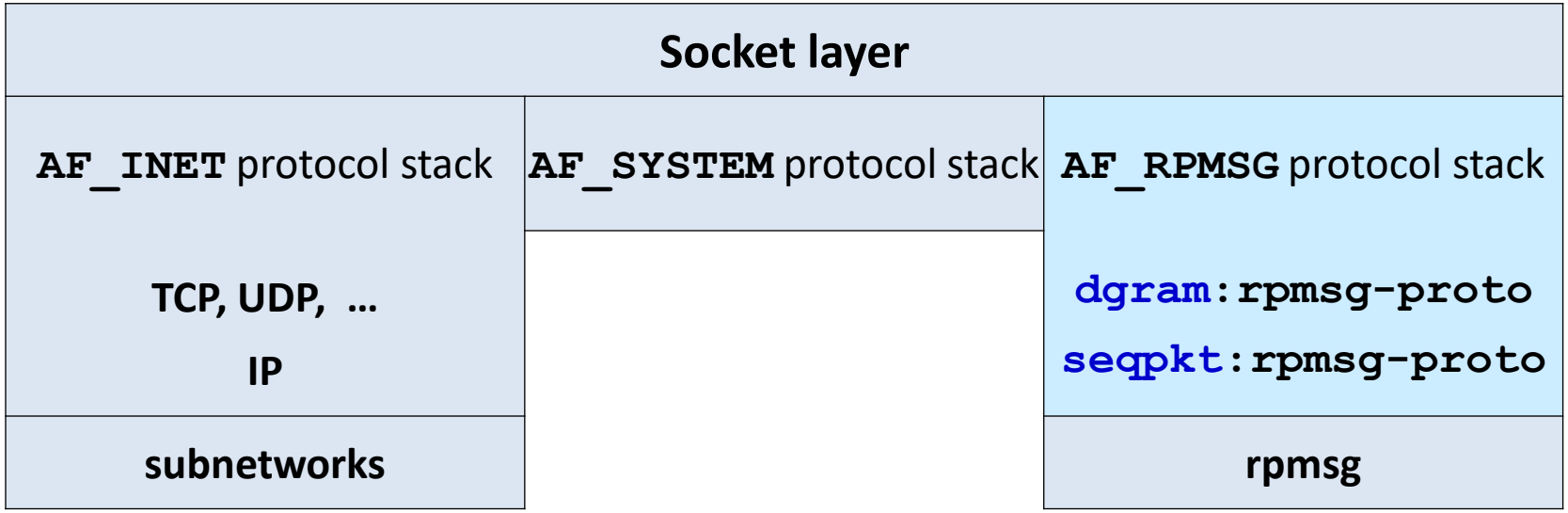
Multiple kernels: examples

- Zynq (*OS asymmetry*)
  - Linux on one Cortex-A9 core as remoteproc master
  - FreeRTOS on the other Cortex-A9 core as remoteproc slave
- Zynq (*Application asymmetry*)
  - FreeRTOS on one Cortex-A9 core as remoteproc master
  - FreeRTOS on the other Cortex-A9 core as remoteproc slave
- Sitara AM572x (*HW platform asymmetry*)
  - Linux on dual core Cortex-A15 MPU as remoteproc master
  - SYSBIOS on each DSP (2x) as remoteproc slave
    (SYSBIOS for DSP is not multicore)
  - SYSBIOS on each dual core Cortex-M4 IPU (2x) as remoteproc slave

IPC & Transport services

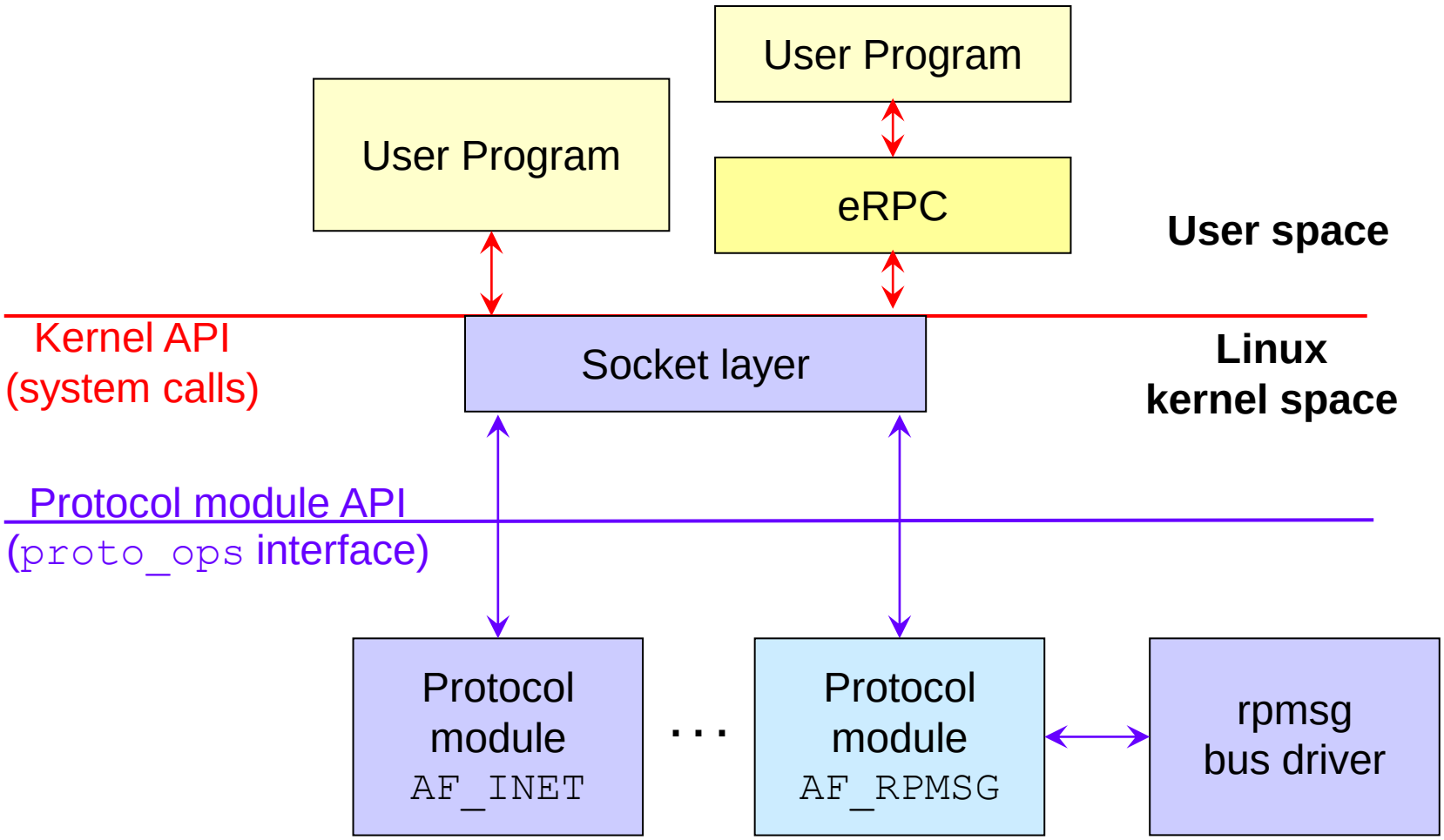
- A de facto standard exists to interface different protocol stacks and multiple transport services: the (BSD) **socket API**
- Different socket types for different transport services
  - **`SOCK_DGRAM:`** message based, unreliable (best effort), connectionless
  - **`SOCK_STREAM:`** stream based, reliable, connection oriented
  - **`SOCK_SEQPACKET:`** message based, reliable, connection oriented
- We offer rpmsg based communication services via the socket API and we implement them as a protocol family: **`AF_RPMSG`**

Socket API & Transport service




```
struct sockaddr_rpmsg {
    short           srpmsg_family;   // AF_RPMSG
    unsigned short  srpmsg_port;
    unsigned long   srpmsg_addr;
    char            srpmsg_zero[8];
};
```

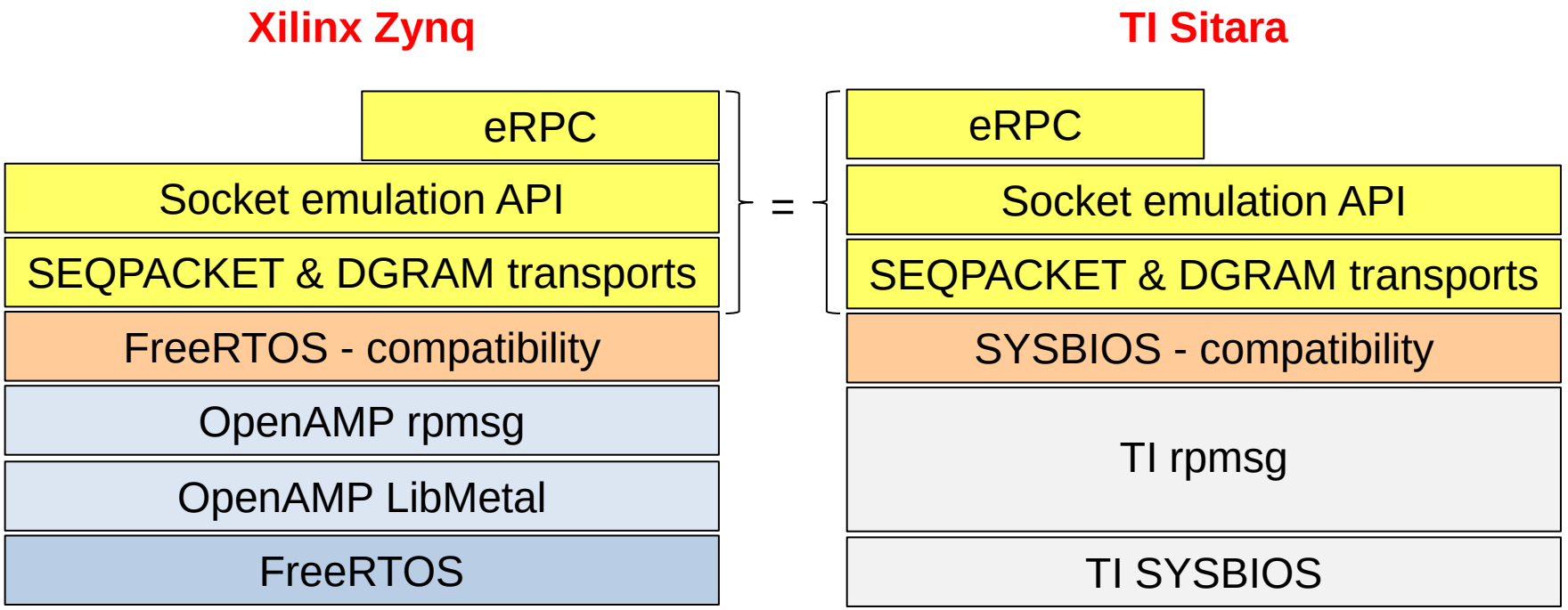
rpmsg transports as a protocol module (and a client driver)



rpmsg on RTOSs

- Focus on FreeRTOS but other RTOSs may be relevant
  - SYSBIOS for TI chips
  - MQX for NXP chips
- Portable implementation provided by OpenAMP
  - Associated to a LibMetal portability library
  - Port available for FreeRTOS Xilinx Zynq/NXP i.MX
  - Only rpmsg bus
  - No link with upstream Linux community
    (port of bus driver on Linux is in user space!)

rpmsg on RTOSs



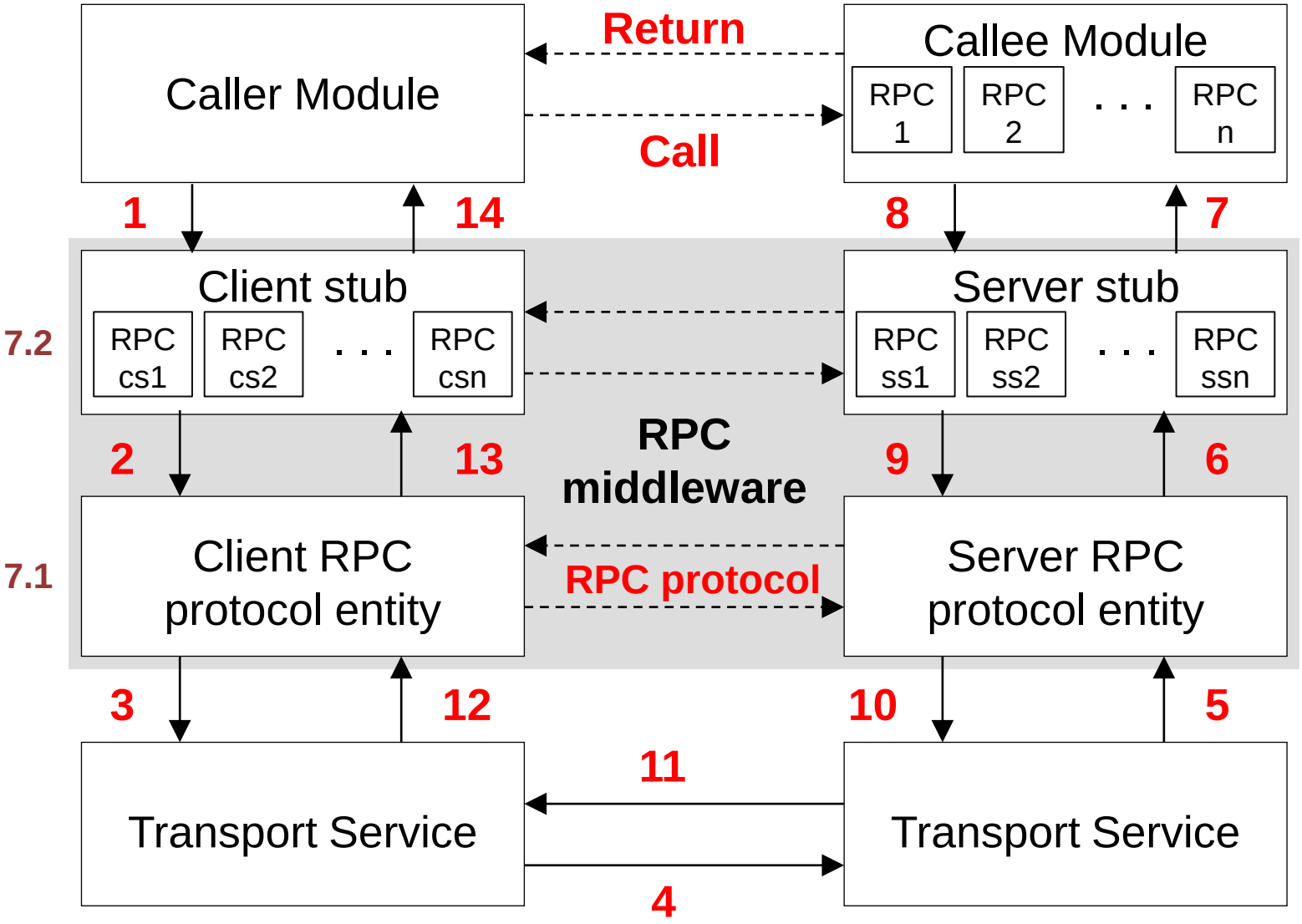
- The RTOS-compatibility layer
  - Provides a uniform client API to the rpmsg layer
  - Provides a uniform API for the access to system resources (time, memory, threads, sync mechanisms)

Programming model

- Pthreads
  - E.g. manager-worker model:
    manager on Linux, workers on RTOSs
  - `pthread_attr_setaffinity_np()` allows to control on what core a thread is created/run
  - How can we share data between threads working on different cores?
- OpenMP

➢ **RPC** (**R**emote **P**rocedure **C**all)

RPC interaction



eRPC .1

- A Remote Procedure Call (RPC) system created by NXP
  - Library
    - RPC protocol entity (Layer 7.1)
    - Un/marshaling of primitive types and type constructors
  - Compiler
    - Automatic generation of stubs (Layer 7.2) from **IDL** (Interface Definition Language)
- Based on portable, passive, OS-independent source code
  - ➢ Main program must be written by the user
  - ➢ A sequential server is assumed
  - ➢ An RPC service is identified by its interface name in its IDL definition

eRPC .2

- Transport independent
- In the original implementation the RPC server must be addressed through a well known address
  - ➢ A client must know a priori where the server is running
- Semantics of an RPC: exactly once
  - ➢ Robustness depends on underlying transport service
- Two call models supported (specified in RPC definition)
  - Returnless (one-way, non confirmed) RPCs
  - Blocking (two-way, confirmed/synchronous) RPCs

eRPC: what was missing

- Mapping on rpmsg transports
- Multiple clients for a single server, and vice versa
- A broker that mediates between clients and servers
  - Allows to locate the provider of an RPC service on the network
    - At startup a server registers itself with the broker (mapping RPC service name → `rpmsg_sockaddr`)
    - When a client wants to use an RPC service it asks the broker “who is providing it?”
  - Global broker for an rpmsg communication domain on the remoteproc master
  - The broker offers its service on the DGRAM transport

Passing images between cores

- Through memory external to the memory that is handled by the Linux kernel (**and** by the other kernels on the chip)
- External memory is “allocated” in the physical address space
- Can be handled within Linux through the **CMEM module**
  - The external memory is described to CMEM in the Device Tree
  - CMEM offers an API in user space
  - CMEM allows to handle physically contiguous chunks of external memory both as cached or as non-cached
  - Shared buffers must be de/allocated through CMEM (by the Linux kernel; remotes may only read/write them)
- Images are passed between cores by passing their physical address

Handling image buffers on RTOSs

- SYSBIOS provides 2 functions to translate back and forth between physical and virtual addresses:
  - `Int Resource_physToVirt(UInt32 pa, UInt32 *da);`
  - `Int Resource_virtToPhys(UInt32 da, UInt32 *pa);`
- The same functions have been implemented on FreeRTOS.

eRPC: performance

- Remote call/return overhead
  - Calling a doNothing() procedure without input and output parameters (TI Sitara AM572x, RPC from MPU to DSP, DGRAM)
  - min = 0.116957ms
  - max = 7.772539ms
  - avg = 0.129622ms
- Image processing operators of the TI image processing library on DSP (RPC on DSP, DGRAM, 1000x738p images)
  - MEDIAN FILTER (3x3): 58.639ms
  - GAUSSIAN FILTER (7x7): 715.661ms
  - EDGE DETECTION: 147.128ms
  - HISTOGRAM CALCULATION: 28.119ms
  - BINARIZATION: 33.212ms
  - MORPHOLOGY (3x3): ~25ms

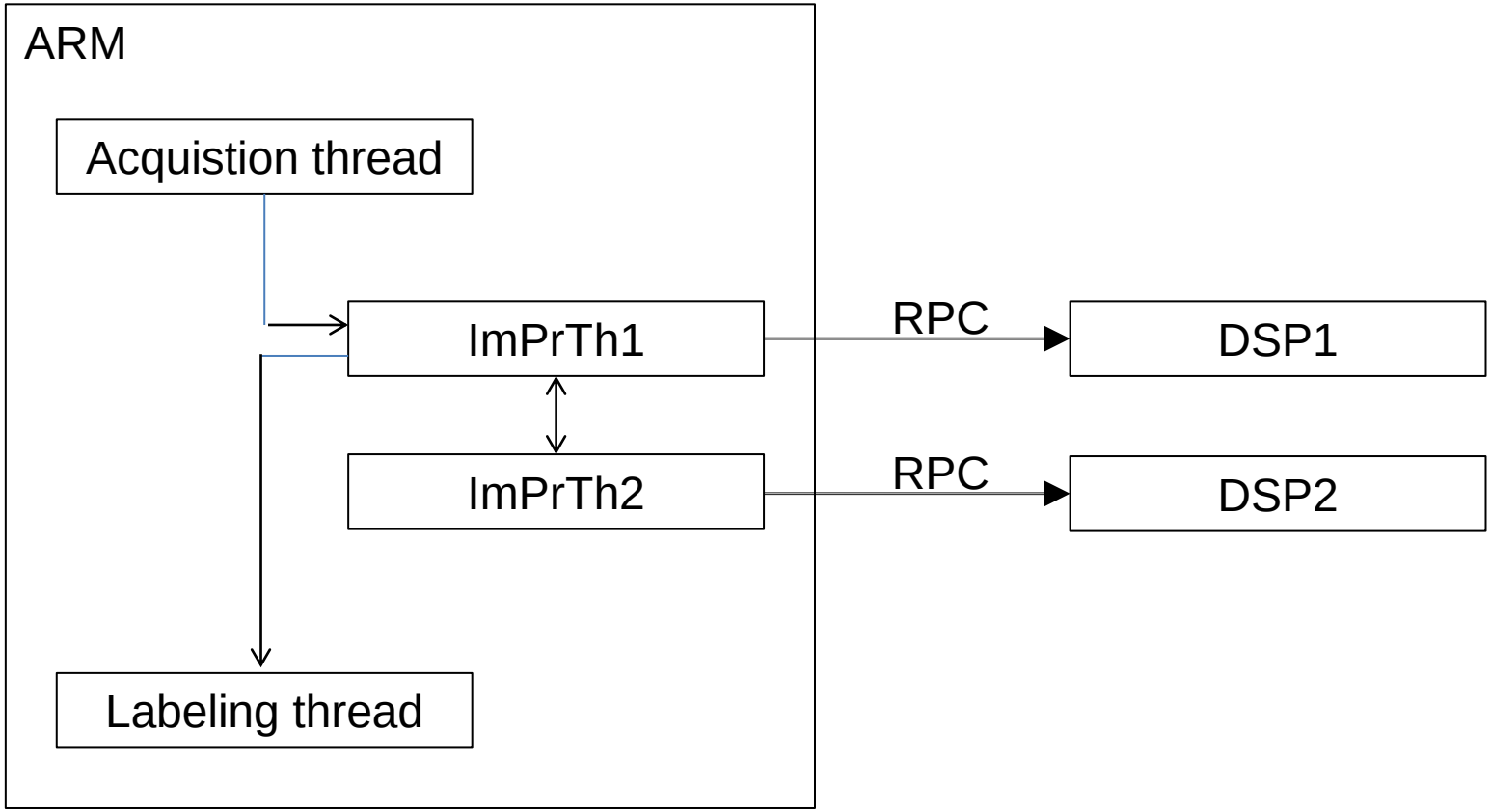
Demonstrator: blob creation and labeling .1

- A 3 stage pipeline runs on the ARM:
  - Image acquisition
    - Implemented as a single thread
  - Image processing pipeline and blob creation
    - Implemented as 2 threads (ImPrTh1 and ImPrTh2) that work in parallel in order to exploit the DSP parallelism
    - One of the two threads (ImPrTh1) has a primary role since it coordinates (when necessary) the work of the other and performs the workk that is actually performed on the ARM
    - ImPrTh1 processes the top middle of the image and delegates work to DSP1
    - ImPrTh2 processes the bottom middle of the image and delegates work to DSP2
  - Blob labeling and display of results
    - Implemented as a single thread

Demonstrator: blob creation and labeling .2

- Image processing pipeline:
  - 3x3 median filter
  - 7x7 Gaussian filter
  - Histogram
    - Partial results merged by ImPrTh1
  - Binarization
  - Compression (1 pixel encoded as 1 byte → 1 pixel encoded as 1 bit)
    - Performed by ImPrTh1
  - Morphology1 .. Morphology5
    - 5 3x3 binary morphology steps
    - Library assumes compressed encoding of images
  - Uncompression (1 pixel encoded as 1 bit → 1 pixel encoded as 1 byte)
    - Performed by ImPrTh1

Demonstrator: blob creation and labeling .3



- Execution time of image processing filters divided by 2 thanks to DSP parallelism
- Frame rate ≅ 2fps

FPGA accelerated Convolutional Neural Networks

T3LAB's work on FPGA based CNN acceleration is run in cooperation with

- the research group of Prof. L. Benini, DEI, Università di Bologna
- the research group of Prof. P. Meloni, DIEE, Università di Cagliari

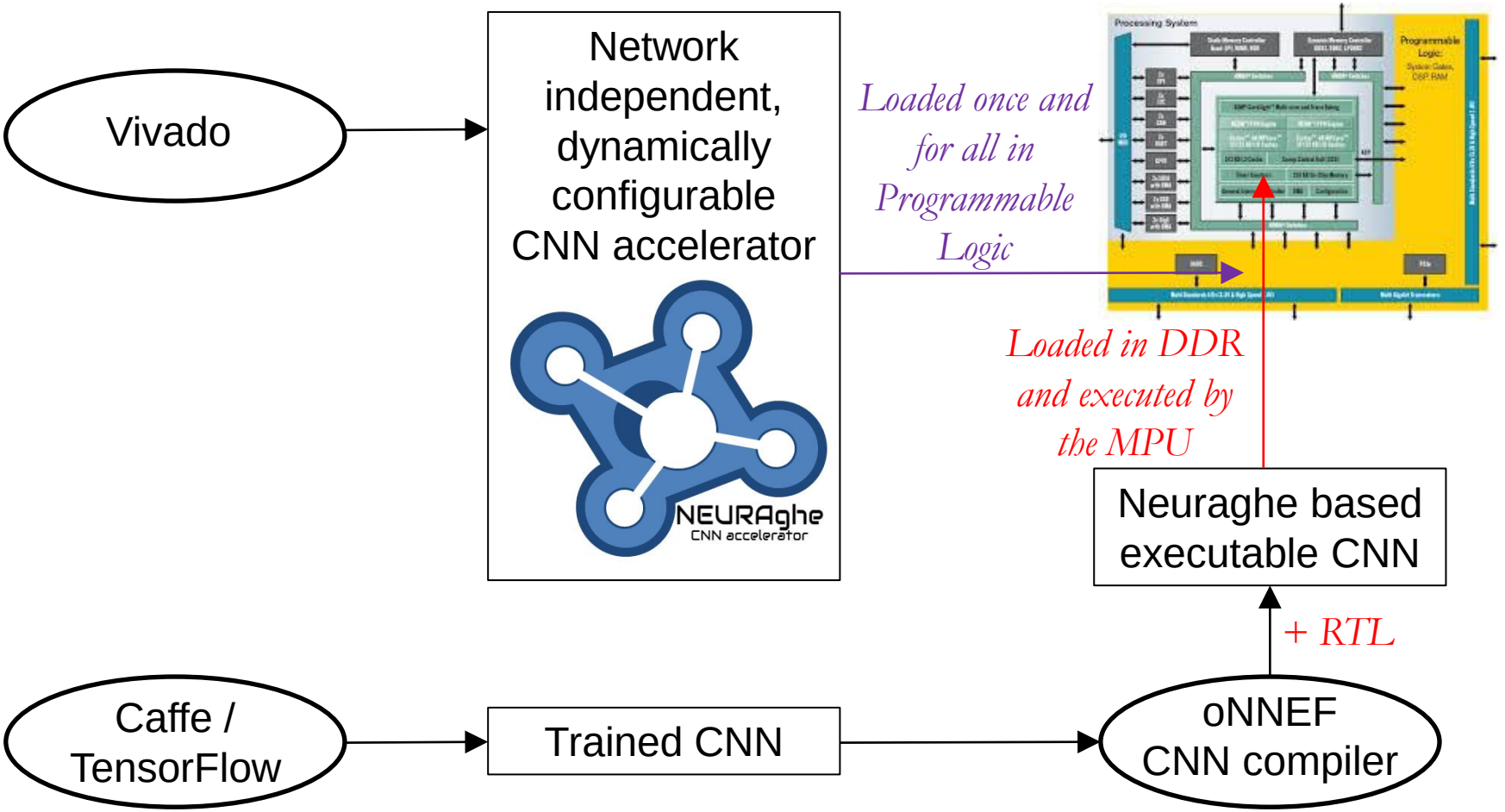
Goals and non Goals

- Goals
  - Accelerate CNN inference using FPGAs in embedded applications
    - The CNN accelerator should be independent from the specific application (CNN), so that it can be loaded once and for all to Programmable Logic
  - Provide a convenient (usability, efficiency) programming environment for the development of FPGA accelerated CNN applications
- Non goals
  - Support the design of CNNs
  - Support the training of CNNs

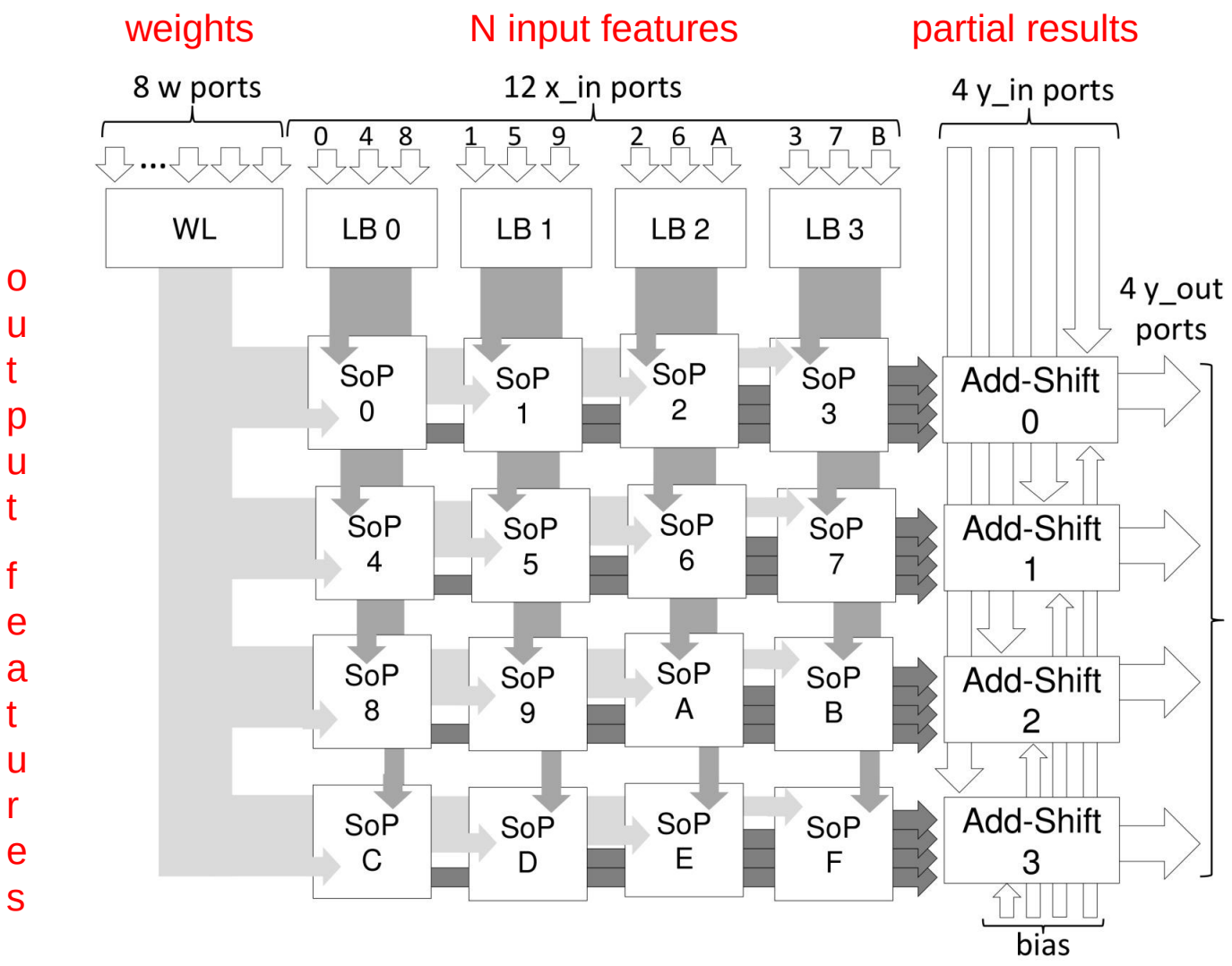
How should it all work together

1. The CNN accelerator (**Neuraghe**) is loaded once and for all to Programmable Logic
2. The CNN is designed and trained using a conventional machine learning framework (e.g. Caffe)
3. Source C code (and a companion weights file) that runs the trained CNN and that takes advantage of the accelerator is generated by the CNN compiler
4. The application is generated by linking the generated source code to the run-time library (RTL) that interfaces the accelerator (and completes the CNN virtual machine)
5. The application is loaded and run on the MPU, thus the CNN is executed on the MPU **and** the accelerator

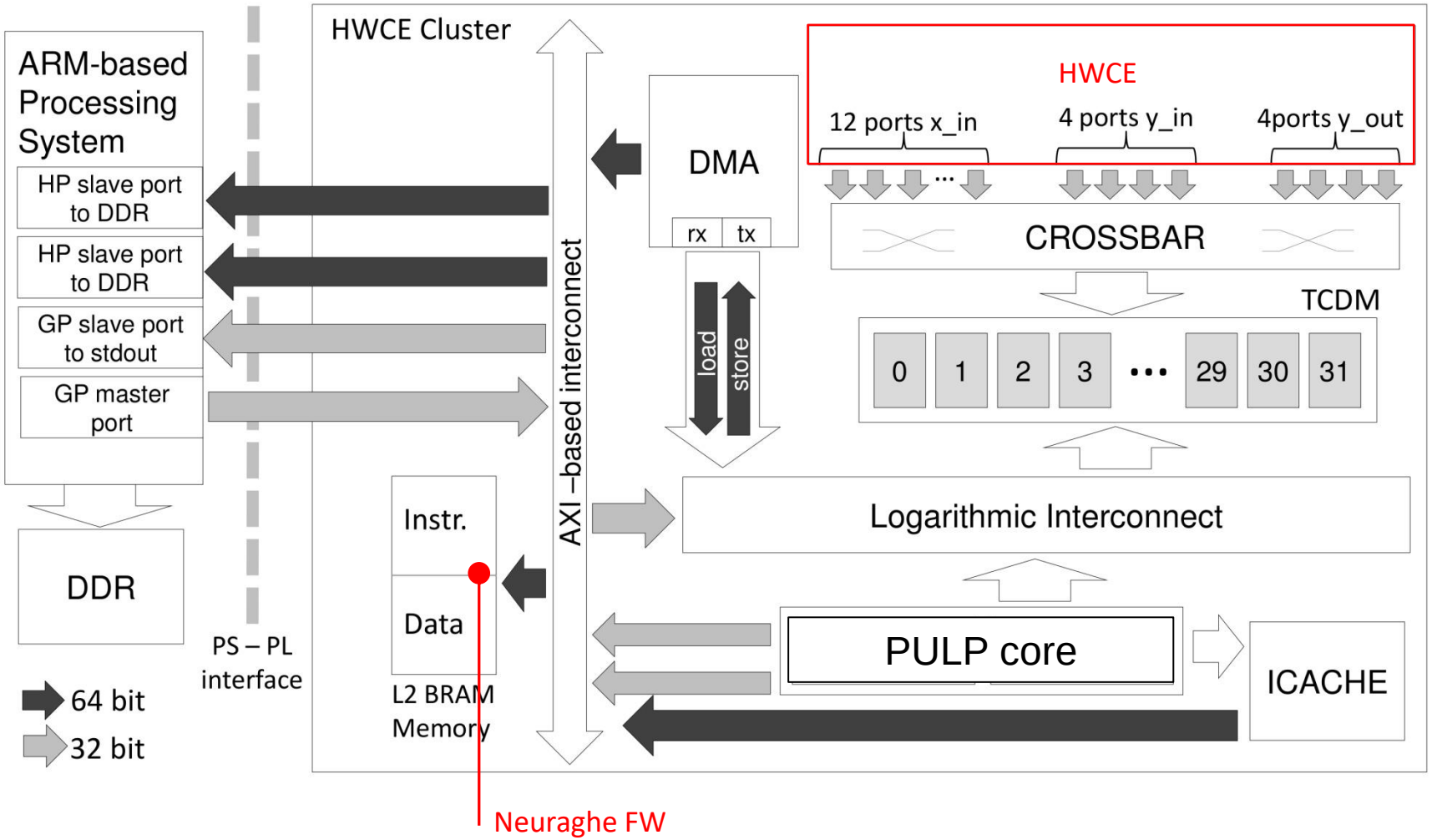
How should it all work together



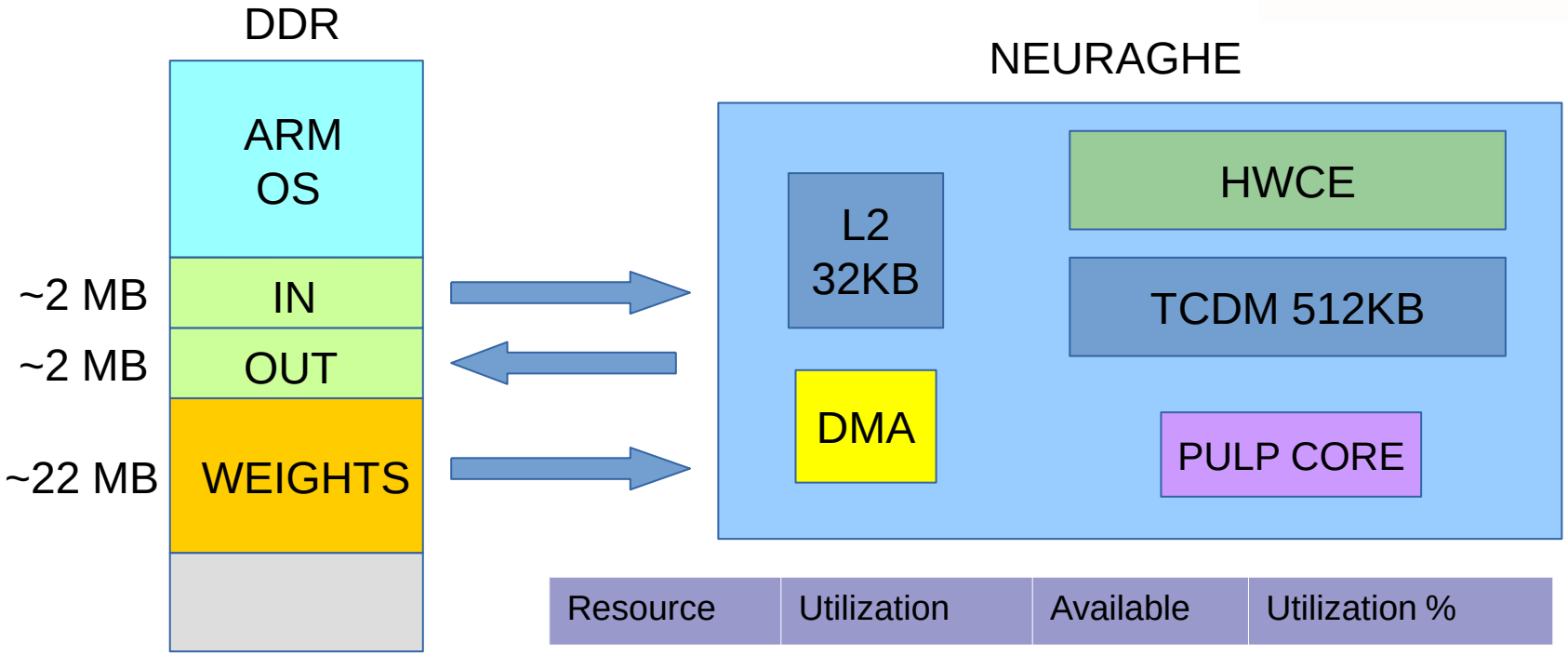
Neuraghe's convolution engine (HWCE)



The complete Neuraghe accelerator



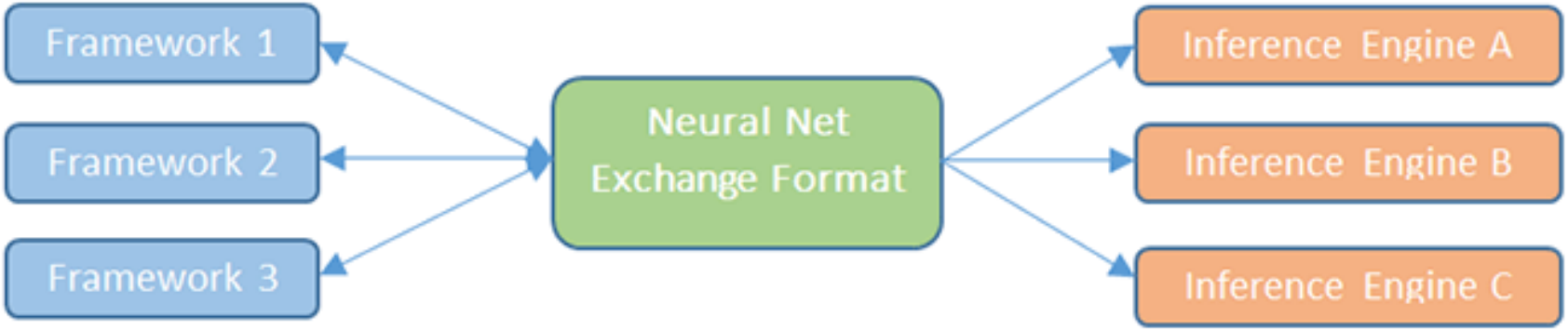
Neuraghe resource needs



On a Xilinx Zynq 7045

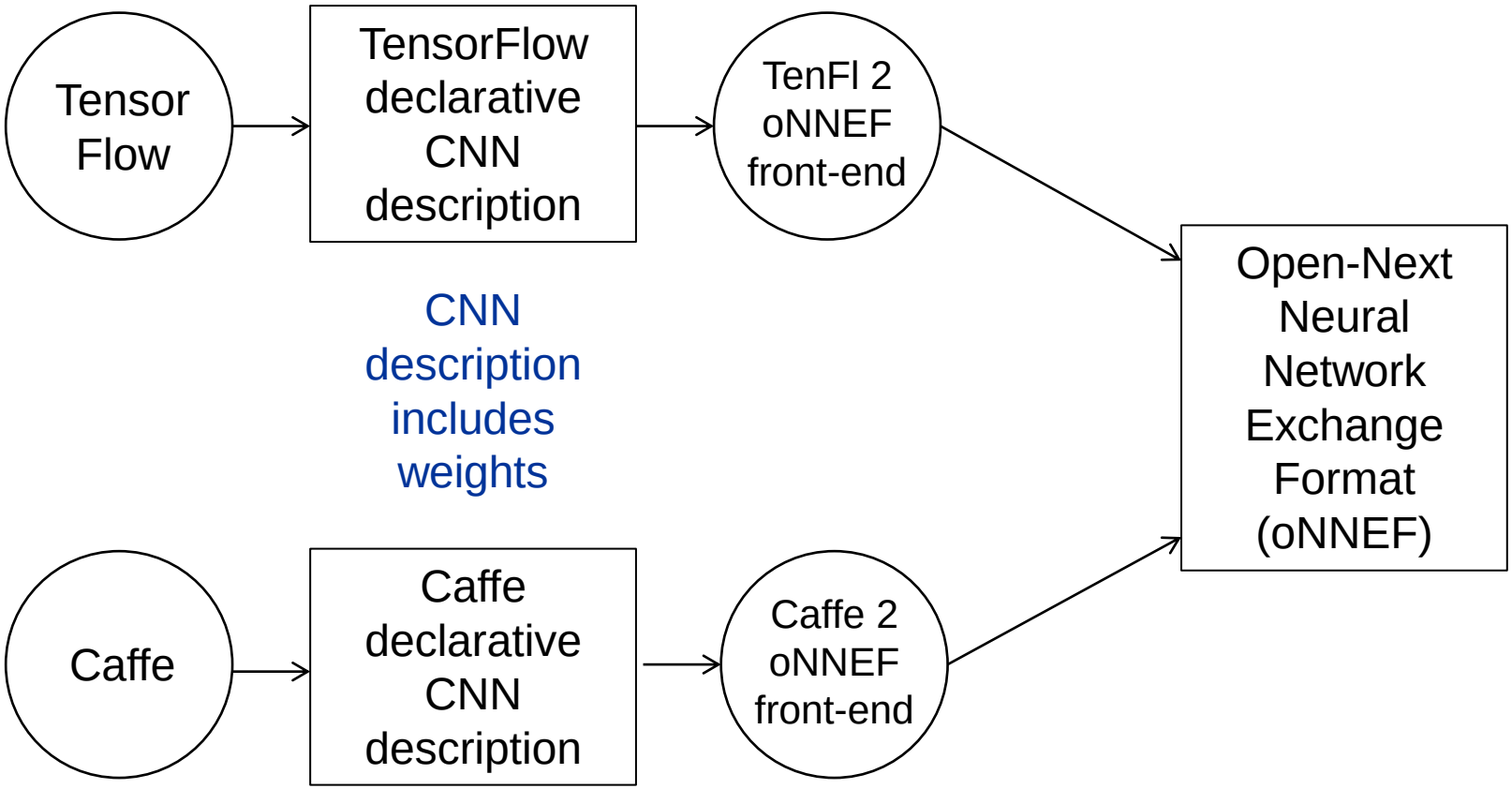
Resource	Utilization	Available	Utilization %
LUT	76729	218600	35.10
LUTRAM	9971	70400	14.16
FF	57422	437200	13.13
BRAM	336	545	61.65
DSP	869	900	96.56
IO	1	362	0.27

Khronos NNEF idea

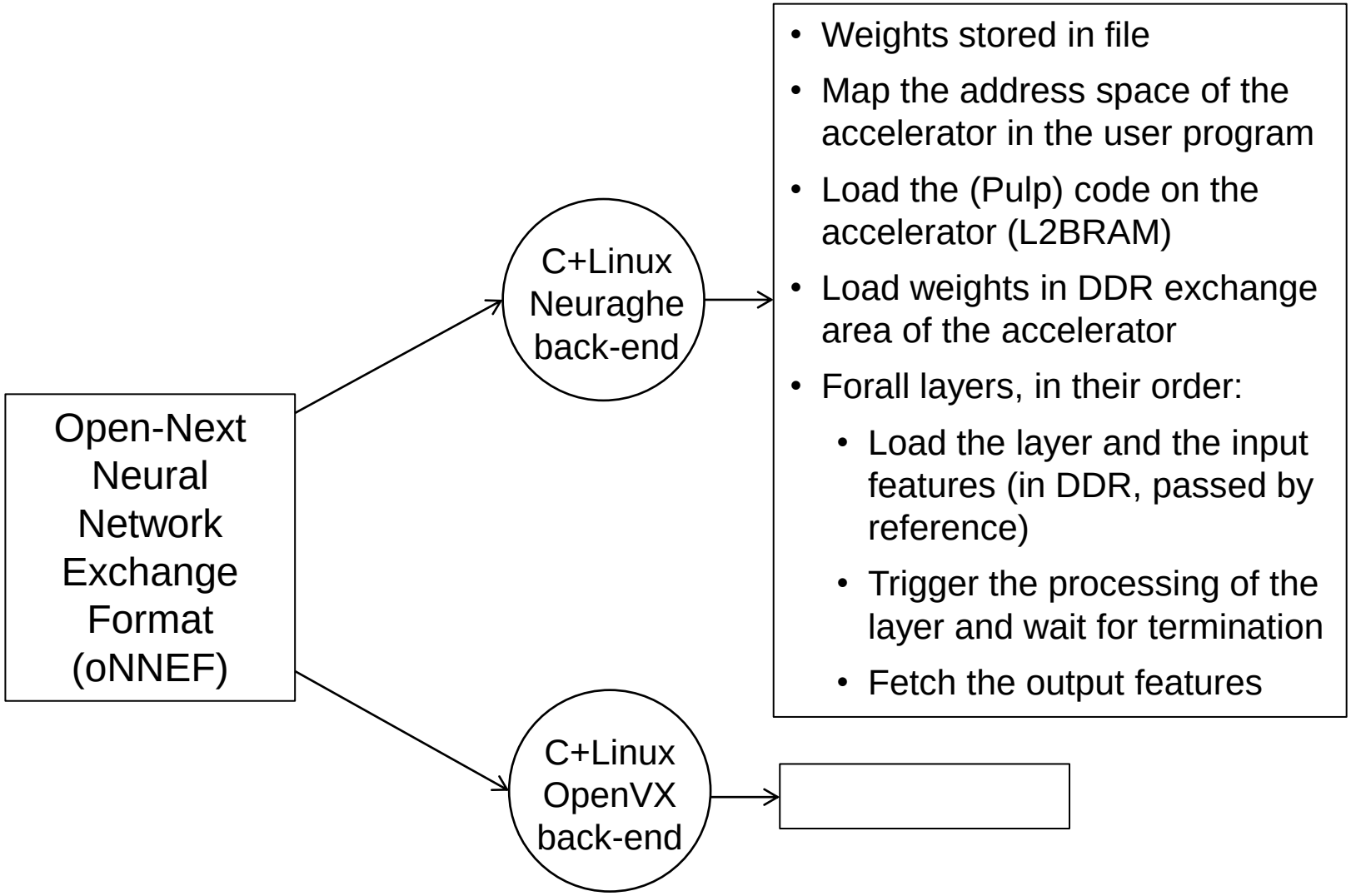


- NNEF = Neural Network Exchange Format
- NNEF not yet available
- oNNEF = Open-Next Neural Network Exchange Format

Programming environment front-end



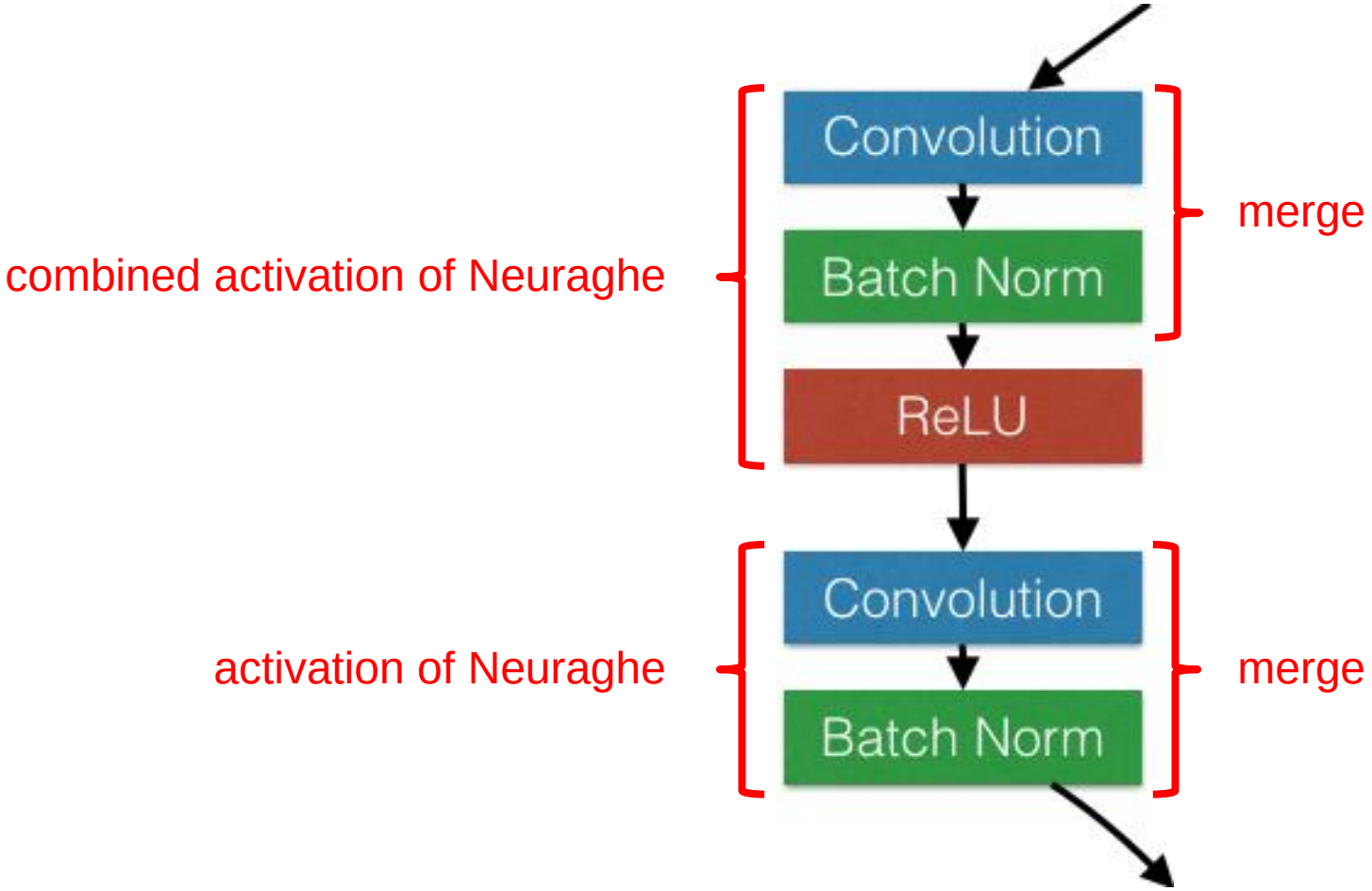
Programming environment back-end



oNNEF Neuraghe compiler characteristics .4

t3lab TECHNOLOGY TRANSFER TEAM

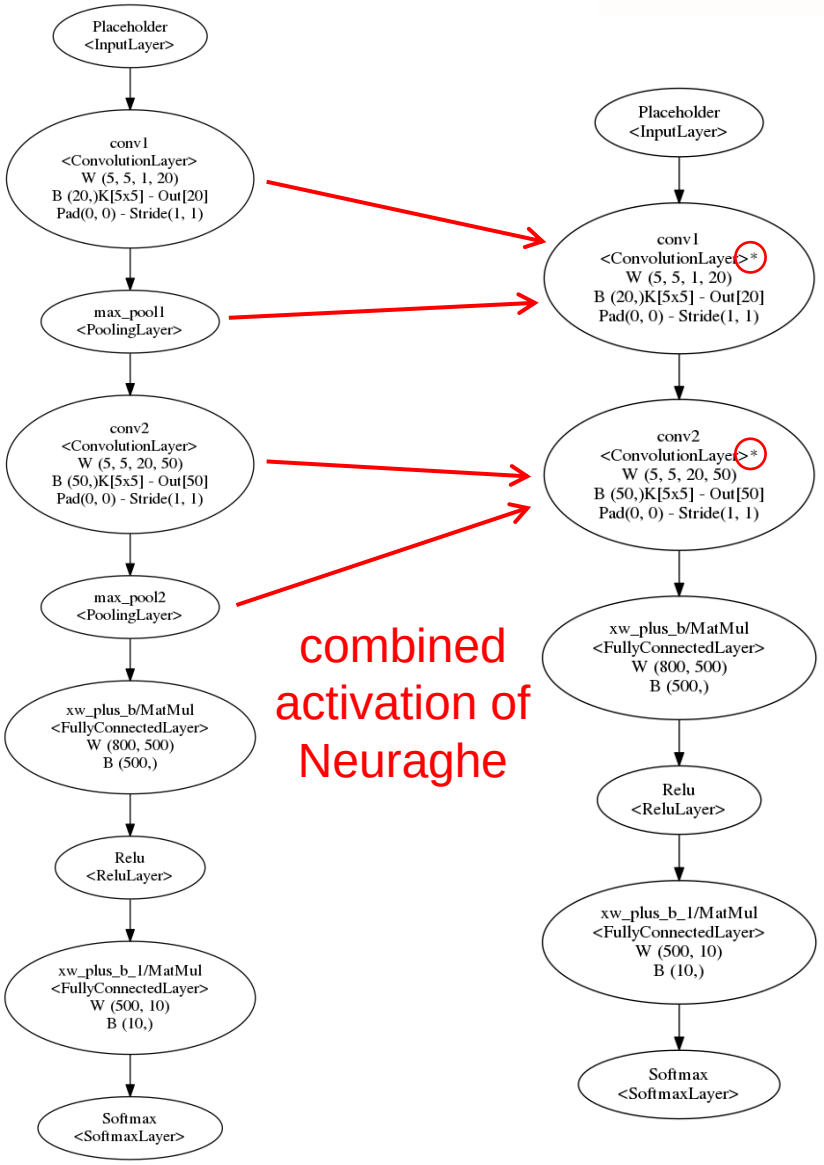
Example 1: optimizations in a section of a ResNet residual block



oNNEF Neuraghe compiler characteristics .5

Example 2:

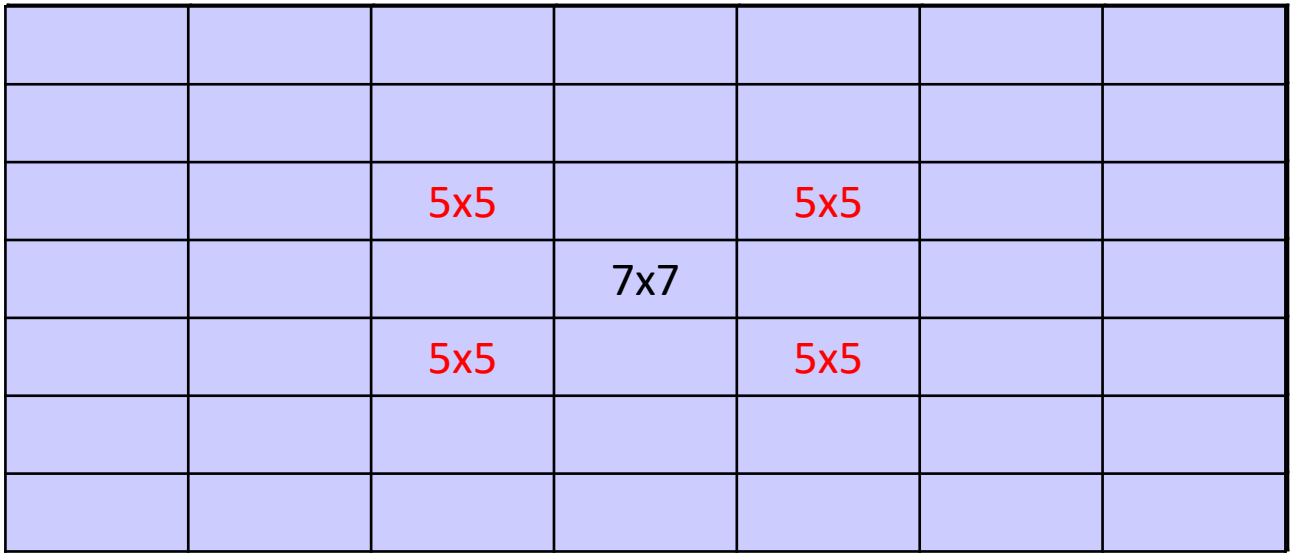
optimization of LeNet



oNNEF Neuraghe compiler characteristics .6

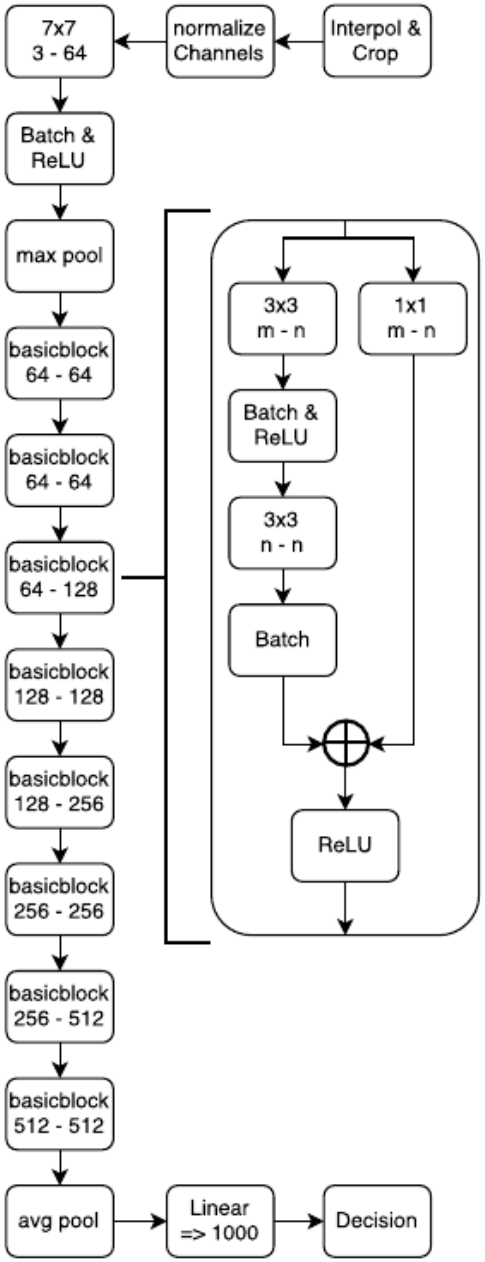
Example 3:

- a 7x7 convolution computed through the recombination of 4 5x5 convolutions
- joint work of the oNNEF compiler and the Neuraghe based CNN virtual machine



ResNet-18

- Consists mainly of the standard parts of a CNN
  - Convolution layers
  - Batch normalizations
  - ReLU
  - Pooling
- A ResNet block consists of two 3x3 convolutions with attached BatchNorm / ReLU
- Each ResNet block has a bypass
- Resulting architecture:
  - 16 3x3 convolution layers
  - 3 1x1 convolution layers
  - 1 7x7 convolution layer
  - 1 fully connected layer (1000 output classes)



On the bypass path there is either a 1x1 convolution or a simple shortcut

A sum-of-features operator is also required

Neuraghe+RTL accuracy

- Work has been done using off-the-shelf (pre-trained) CNNs:
  - LeNet
  - ResNet-18
- Accuracy achieved by Neuraghe+RTL has been tested against the results computed (using floating-point arithmetic) by the original machine learning framework
- Comparison based only on ranking of results

➢ For LeNet there is a perfect match between floating-point and fixed-point results

➢ For ResNet-18 the match is only partial (1960 test images):

- Good match: one of the two best fixed-point results matches the best floating-point result: 67.2%
- Partial match: at least one of the best 5 fixed-point results matches one of the best 5 floating-point results: 24.3%
- No match: none of the above: 8.5%

Neuraghe+RTL performances .1

- Dual core ARM frequency: 667MHz
- Neuraghe:
  - PULP frequency: 50MHz
  - HWCE frequency: 110MHz

ResNet-18 (224x224 pixel images) execution time (ms)	
Neuraghe on Zynq	~260
4-core i7-6700 @3.4GHz	> 300
Zynq (single core, no NEON) @667MHz	~58000

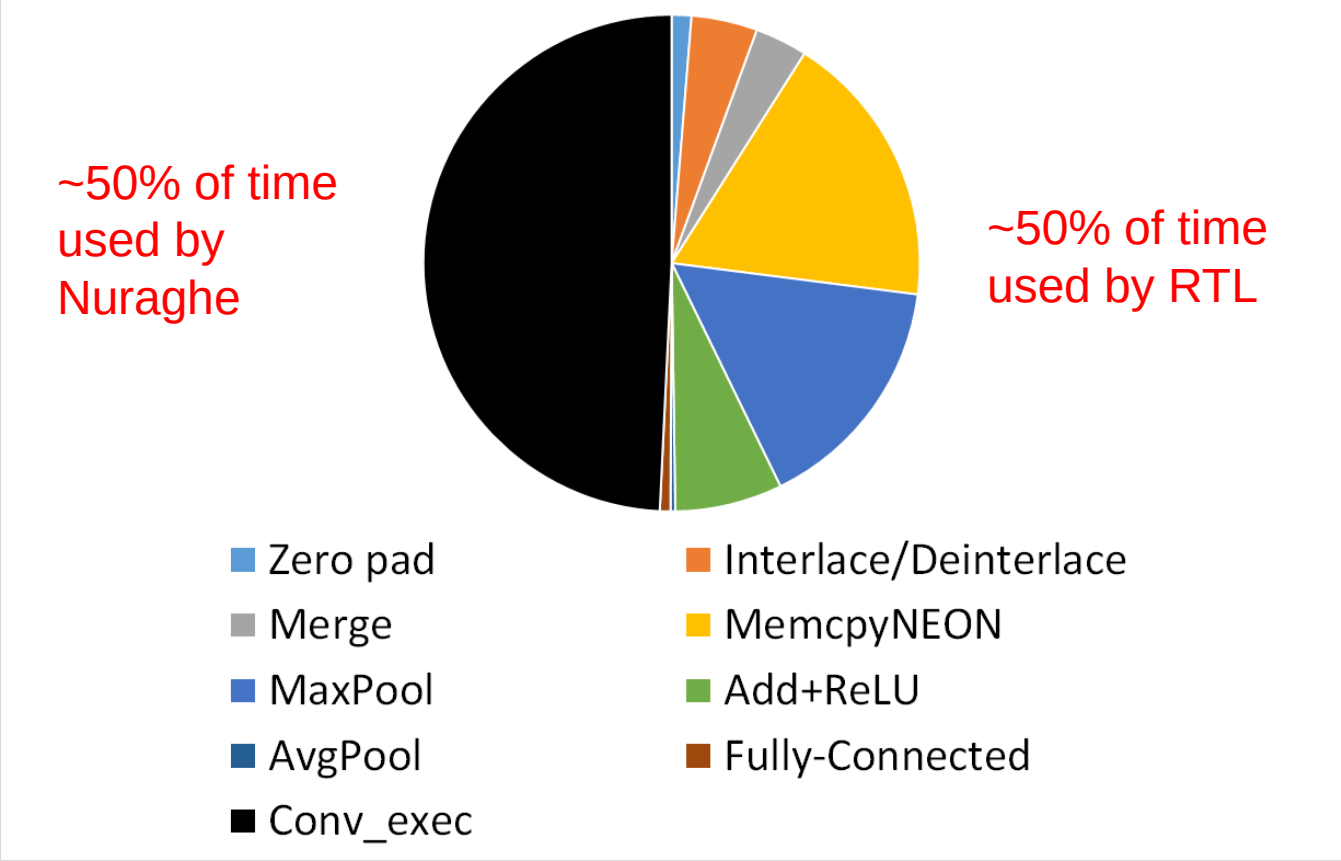
Neuraghe+RTL performances .2

- Dual core ARM frequency: 667MHz
- Neuraghe:
  - PULP frequency: 50MHz
  - HWCE frequency: 100MHz

LeNet (28x28pixel images) execution time (ms)	
Neuraghe on Zynq	~2.1
4-core i7-6700 @3.4GHz	1 .. 2.7

Neuraghe+RTL performances .3

- ResNet-18 (224x224 pixel images)
- Dual core ARM frequency: 667MHz
- Neuraghe: PULP frequency: 50MHz / HWCE frequency: 110MHz



Acknowledgements

- The OPEN-NEXT project is co-funded by

  thanks to POR FESR funds

- michele.brian@t3lab.it
  davide.berardi@t3lab.it
  marco.melletti@t3lab.it
  alessio.paccoia@t3lab.it
  michele.rodolfi@t3lab.it
  claudio.salati@t3lab.it
  matteo.sartori@t3lab.it