software framework for runtime-Adaptive and secure deep Learning On Heterogeneous Architectures

Project Number 780788

Project Acronym ALOHA

D4.1	Report on hardware abstraction layer techniques		
Work Package:	WP4	**Lead Beneficiary:**	ST-I
Type:	Report	**Dissemination level:**	Public
Due Date:	30th September 2018	**Delivery:**	30th September 2018
Version:	D4.1_Report on hardware abstraction layer techniques_v1.0.doc		

Brief description:

The purpose of this deliverable is to report on the research and development activities concerning support for hardware abstraction layer and runtime environment for the ALOHA reference computing platforms.

Deliverable Author(s):

Name	Beneficiary
Giulio Urlini	ST-I
Paolo Meloni	UNICA
Nikos Fragoulis	IL

Deliverable Revision History:

Reviewer Beneficiary	Issue Date	Version	Comments
ST-I	02/08/2018	v0.1	Released of the Table of Content and introduction
ST-I	10/09/2018	v0.2	Added section 3.2
IL	26/09/2018	v0.3	Added contributions on Chapter 3
UNICA	27/09/2018	v0.4	Final draft
UNICA	28/09/2018	v0.5	Incorporated reviews from REPLY and CA
ST-I	30/09/2018	V1.0	Final review

Disclaimer

The ALOHA Consortium is the following:

#	Participant Legal Name	Acronym	Country
1	STMICROELECTRONICS SRL	ST-I	Italy
2	UNIVERSITA' DEGLI STUDI DI CAGLIARI	UNICA	Italy
3	UNIVERSITEIT VAN AMSTERDAM	UVA	Netherlands
4	UNIVERSITEIT LEIDEN	UL	Netherlands
5	EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZUERICH	ETHZ	Switzerland
6	UNIVERSITA' DEGLI STUDI DI SASSARI	UNISS	Italy
7	PKE ELECTRONICS AG	PKE	Austria
8	CA TECHNOLOGIES DEVELOPMENT SPAIN SA	CA	Spain
9	SOFTWARE COMPETENCE CENTER HAGENBERG GMBH	SCCH	Austria
10	SANTER REPLY SPA	REPLY	Italy
11	IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD	IBM	Israel
12	SYSTHMATA YPOLOGISTIKIS ORASHS IRIDA LABS AE	IL	Greece
13	PLURIBUS ONE SRL	P-ONE	Italy
14	MAXQ ARTIFICIAL INTELLIGENCE, LTD (formerly MEDYMATCH TECHNOLOGY, LTD)	MaxQ-AI (formerly MM)	Israel

Table of Contents

Figures

Tables

1 Executive Summary

The main objective of this deliverable is to describe the activities performed from M4 to M9 regarding the automatically implementation of the partioning and mapping of the algorithm configuration on the ALOHA reference computing platforms.

The deliverable describes the progress made on the translation of the partitioning and mapping description into a platform-specific code, starting from the definition of an architecture description format (Section 2) to the collection of information on the computing resources and programming primitives available for each target reference architecture (Section 3).

This report relates to the following future deliverables:

- D4.2, which will update the current content to reflect the latest developments at M18;
- D4.3 and D4.4, which will include a first and a final release of the hardware abstraction layer utilities initially outlined in this deliverable.

ST-I, UNICA, ETHZ, IL, REPLY and CA have contributed to this deliverable.

1.1 Acronyms and abbreviations

Acronym	Meaning
WP	Work Package
DL	Deep Learning
AI	Artificial Intelligence
SoC	System on Chip
CNN	Convolutional Neural Network
CSP	Convolution Specific Processor
GPP	General Purpose Processor
CE	Convolution Engine
DMA	Data Memory Access
µC	Microcontroller
TCDM	Tightly-Coupled Data Memory
NeuDNN	NEURAghe Deep Neural Network
HAL	Hardware Abstraction Layer
WM	Weight Memory
WDMA	Weight Data Memory Access
GOPs	Giga Operations Per Second
COTS	Commercial Off-The-Shelf
IDE	Integrated Development Environment
SDK	Software Development Kit
DDR	Double Data Rate
DSP	Digital Signal Processor

2 Automation of target-specific code generation

The steps needed for automating the implementation of the partioning and mapping of the algorithm configuration produced by the upper levels of the ALOHA toolflow (WP2 + WP3) on the target hardware platforms are mainly three:

- abstract the characteristics of the target platform;
- automate the translation of the partitioning and mapping description into a platform-specific code that exploits the programming primitives exposed by the target processing platform, that must be used to execute a processing or communication task on the hardware architecture;
- customize and instrument the code to reduce as much as possible the power consumption of the target hardware, using, when available, power reduction techniques such as power gating, clock gating, frequency scaling and others.

During the period M4-M9, the Consortium was mainly involved in the first two activities. We have developed an architecture description format which is going to be assessed in the months following the release of this deliverable. Then, we have confirmed the use of three main reference platforms to assess the project outcomes:

- NEURAghe[1], a Zynq-based heterogeneous architecture accelerating CNNs released by UNICA and ETHZ;
- Orlando[2], a low power architecture for CNN acceleration developed by ST-I;
- SnapDragon 845[3], a mobile processing platform released by Qualcomm.

For each target architecture, we have conducted an analysis of the computing resources and programming primitives. We have also worked on the APIs exposed by the target platforms to provide more flexibility and to support the automatic use of the programming model. The customization of the code to reduce power consumption is planned for the next period.

2.1 Abstraction of the target platform characteristics

At month M6, a first version of the architecture description format for the target reference platforms has been provided in D1.1 (Report on general specifications and interface definition). The abstraction model provides information on the population of computing elements, connectivity, and available operating modes (data types, working frequency and gating conditions), and describes a set of operators/actors representing the elementary computation and communication tasks that can be triggered on the computing resources exposed by each processing platform. For more information please see section 4.2 of D1.1.

2.2 Translation of the partitioning and mapping description into a platform-specific code

For each target architecture, we have defined the available computing resources and the supported programming models, which are described in more detail in section 3.

[1] P. Meloni et al., "*NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs*", 2017, https://arxiv.org/abs/1712.00994
[2] G. Desoli et al., "*14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems*", 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, 2017, pp. 238-239.
[3] https://www.qualcomm.com/products/sdm845

3 ALOHA target platforms

The ALOHA approach will be practically validated on three main reference platforms, showing that it can actually be supported by state-of-the-art technologies. In this section we describe the main features of the selected target platforms, mainly focusing on the available computing resources and the supported programming models.

3.1 NEURAghe

NEURAghe is a Zynq-based processing platform for CNN, specifically designed to improve flexibility and re-usability in different context and for the implementation of different CNN-based algorithms. It leverages the synergistic usage of Zynq ARM cores and of a powerful and flexible Convolution-Specific Processor (CSP) deployed on the reconfigurable logic. The CSP embeds both a convolution engine and a programmable soft core, releasing the ARM processors from most of the supervision duties and allowing the accelerator to be controlled by software at an ultra-fine granularity.

3.1.1 Available computing resources

NEURAghe is built on top of a Xilinx Zynq SoC. As shown in Figure 1, it leverages both a dual ARM Cortex-A9 processing system, which is used as General-Purpose Processor (GPP), and a reconfigurable logic, which hosts the CSP.

The GPP is used as an active partner in the heterogeneous computation of complex CNN topologies, carrying out tasks that would be accelerated less effectively on the programmable logic, such as memory-bound fully connected layers. The CSP is composed of several submodules:

- a local tightly-coupled data memory (TCDM) used to store activations and runtime data,
- a weight memory (WM),
- a weight DMA controller to move weights to the CSP (WDMA),
- an activation DMA to move activations in/out of the CSP (ADMA),
- a simple microcontroller soft-core (μC),
- a Convolution Engine (CE) that embeds the sum-of-products units used to deploy convolutions on the reconfigurable logic.

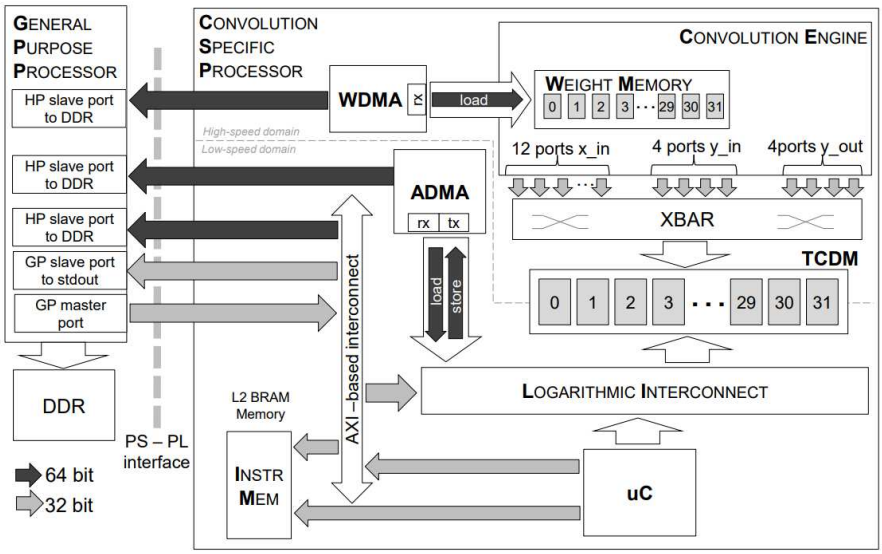

Figure 1: NEURAghe architectural template

NEURAghe exploits two high-performance 64-bit ports for CSP-to-GPP communication (e.g. to access the memory-mapped off-chip DDR) and two general-purpose 32-bit ports for memory-mapped control of the NEURAghe architecture and standard output.

3.1.2 NEURAghe programming model

NEURAghe is provided with an open-source multi-target structured software stack which enables the reuse of existing hardware, software and algorithms on the platform. The NEURAghe Deep Neural Network software stack (NeuDNN) consists of a C/C++ front-end, which can be used to specify and program CNN at software level, and of a back-end that maps processing kernels to the hardware accelerator and controls their execution.

The NeuDNN Front-End is a configurable C/C++ library for CNN deployment. It gives access to a set of statically linkable functions implementing pre-optimized layers and utilities for CNN development with no dependency from third party libraries. The NeuDNN targets efficiently ARM class A processors and the NEURAghe architecture, supporting different activation format data types, such as 32-bit IEEE floating point and 16-bit fixed point. The main NeuDNN computational kernels available as linkable C/C++ API are shown in Table 1.

Table 1: NeuDNN Kernels

Kernel	Dimensions	Stride	Data type	Deploy	Optimization	Note
Convolution	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
*Convolution**	1x1, 3x3, 5x5	4,2,1	16-bit fixed	CSP	CE	Async, sync
Max Pooling	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
*Max Pooling**	2x2, 4x4	4,2,1	16-bit fixed	CSP	CE	After *Convolution**
Avg Pooling	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
*Avg Pooling**	2x2, 4x4	4,2,1	16-bit fixed	CSP	CE	After *Convolution**
Fully-Connected	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
Add	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
ReLU	Arbitrary	-	float, 16-bit fixed	GPP	OpenMP, NEON	
*ReLU**	Arbitrary	-	16-bit fixed	CSP	CE	After *Convolution**
Identity	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
LRN	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	

By default, the library offers optimized implementations for all kernels and the data types deployable to the Generic Purpose Processor. All layers are optimized using OpenMP parallel programming model to exploit parallelisms on the host-side, and ARM NEON vectorization to exploit SIMD acceleration. When CSP is available, some of those layers can be offloaded to the NEURAghe Convolutional Engine. The CSP-based and the GPP-based implementations share the same APIs, thus the library may forward transparently the execution of the layer to most efficient engine. To enable cooperative computation between the host and the CSP, the hardware accelerated *Convolution** layers support blocking and non-blocking semantics. Like software tasks, multiple *Convolution** layers can be enqueued to the accelerator, while the host processor can be used to compute in parallel other layers. These features are enabled by the lower level of NeuDNN software stack.

The NeuDNN back-end – transparent to the user – is composed of a NeuDNN Driver, used to offload computational task to the FPGA-accelerator, and of a Convolution Specific Processor resident RTE, executed by the μC, that receives requests from the driver and schedules autonomously the computation kernels on the Convolutional Engine and data transfers on the DMAs. The driver takes care of the buffer marshaling and of the general transfers between the host DDR partition and the NEURAghe Convolution Specific Processor. Actions on the accelerator are triggered by the driver by means of dedicated commands, consisting in a set of meta-data structures that carry the information needed for the execution of the API

(such as weight array pointers, activation array pointers, etc.). Commands are stored in a shared FIFO queue mapped on the DDR address space. Being NeuDNN implemented on top of the Linux OS, the DDR must be split in two partitions: one used by the OS as main virtual memory; and other one, unmapped and accessed by /dev/mem, contiguous and not paged, used to share data buffers between GPP and CSP. The CSP-side is fully managed by a resident runtime, executed by the µC in the CSP, which is loaded and activated at the startup of the system, just after the load of the bitstream on the programmable logic. The runtime, written in C, has direct access to the CSP HAL and is in charge of orchestrating data transfers from/to the local Convolutional Engine TCDM and triggers of CE activations. The runtime decomposes commands received by the GPP driver, requesting CNN basic operations such as Convolutions, Max Pool layers and ReLUs, into a scheduled track of elementary operations on the CE and on the two DMAs. The used scheduling strategy is aggressively optimized to improve efficiency under limited bandwidth availability, using double-buffering and sliding window techniques to optimize the overlapping of computation with data transfers.

To implement a CNN, a user must develop a C/C++ code, exploiting NeuDNN APIs, and must define a simple configuration file describing the target computing platform (for example ARM SoC, or NEURAghe). To load the data needed for the inference (weights and bias values), the user can rely on some migration tools provided by the NeuDNN to easily import trained models from common ML tools, such as Tensorflow[4] and Caffe[5].

As an example of the actual implementation of the actors listed in Table 1, we report a list of the primitive functions that may be used to trigger execution of different CNN actors on the CSP in NEURAghe.

```
/*
*
* NEURAghe API
*
*
*/


int convolution_hw(SPATCONV sc, DATA* input, DATA* output, SOCMAP soc, SIZE in_s[3], SIZE
out_s[3], SIZE stride[2], SIZE pad[2], bool activate, int qf);
      //Executes a hardware convolution.
            //activate: enables hardware RELU
            //qf:       sets the qf value


RET convolution_wait(SOCMAP soc, int job_id);
      //Busy wait for the end of the hardware convolution


// FULLY CONNECTED layer
// FC is performed by ARM core

LINEAR_FP16 fully_connected_create(void);
      //Creates a new structure for the FC

RET fully_connected_init(LINEAR_FP16 lin, NAME weightsFile, NAME biasFile, SIZE in_s, SIZE
out_s);
      //Popolates the field of the FC structure
```

[4] MartÃŋn Abadi et al. "*TensorFlow: A system for large-scale machine learning*", 2016. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI '16), 265 – 283.
[5] Yangqing Jia et al. "*Caffe: Convolutional architecture for fast feature embedding*". 2014. In Proceedings of the 22nd ACM international conference on Multimedia (MM '14), 675 – 678.

```
RET fully_connected(LINEAR_FP16 lin, int16_t* input, int16_t* output, SIZE in_s, SIZE
out_s);
      // Excecutes the FC on the given input

RET fully_connected_destroy(LINEAR_FP16 lin);
      //Frees the memory

// AVERAGE POOLING
// Performed by ARM core
RET avgpool(int16_t* input, int16_t* output, SIZE in_s[3], SIZE out_s[3], SIZE kern_s[2],
SIZE stride[2], SIZE pad[2], int qf);
      //Given the the input activations returns a single value for each feature map.


// MAX POOLING
// Performed by ARM core
RET maxpool(int16_t* input, int16_t* output, SIZE in_s[3], SIZE out_s[3], SIZE kern_s[2],
SIZE stride[2], SIZE pad[2]);
      //This funcion makes the subsampling of the activations with the maxpooling


//MEMORY COPY
void *memcpyNEON(void *dst, const void *src, size_t len);
      //When NEON is supported and enabled this function performs the copy of a memory area


// Low-level HAL

int * addr_linux(neuADDR addr, int size, int fd);
      // Given a physical address, a size and the pointer to /dev/mem returns the mapped
virtual address

void init_soc(SOCMAP* socs, DATA** wPointer, int MAXMEM, int SWAPMEM, char* bitstream_file);
      // loads the bitstream into the programmable logic and initialiaze the soc

int mmap_soc (SOCMAP* socs);
      // populates the structure with the virtual addresses

int munmap_soc(SOCMAP* socs);
      // frees the memory

int load_bitstream(char* bitstream_file);
      // loads the bitstream into the PL

int bitstream_check(void);
      // check if the bitstream has been already loaded

int conv_setup (volatile Conv_params * task_ptr, int in_f, int out_f, int ih, int iw, int
fs, int max_og, int rectifier_activ, int pooling, int qf, int zero_padding, int w_addr, int
x_addr, int y_addr);
      // low level API to offload a convolution to the accelerator. It is called by high
level API

int conv_hwce(volatile int * ddr_addr, volatile Conv_params * task_ptr, DATA *W, DATA *x,
int in_depth, int out_depth, int in_height, int in_width, int fs, int maxog, int activation,
int pooling, int qf, int precision8, int zero_padding, DATA *y);
      // low level API to offload a convolution to the accelerator. It is called by high
level API

void load_instr (volatile int * soc_addr);
      // writes the instructions for the RISC-V soft processor

void fetch_enable (volatile int * soc_cntr_addr);
      // enables the executions of the instructions by the soft processor

void lock_ps_regs(int* ps7_slcr_addr);
      // locks the reserved PS7 registers
```

```
void unlock_ps_regs (int* ps7_slcr_addr);
      // unlocks the reserved PS7 registers

void set_fclk_div(int* ps7_slcr_addr,int div1, int div2);
      // Sets the divisors to change the frequency of the PL's clocks

void start_fclk(int* ps7_slcr_addr);
      //starts the PL clocks

void stop_fclk(int* ps7_slcr_addr);
      //stops the PL clocks


void wait_for_conv(volatile int * soc_cntr_addr, volatile int * soc_addr);
      // low-level busy wait for the hw convolution

void wait_for_mw_ready(volatile int * ddr_addr);
      // after the fetch_enable ARM must wait for the bootstrap of the soft processor
```

3.2 Orlando

The Orlando platform is presented again here in order to introduce the programming model that can allow the access to such platform.

The STMicroelectronics Orlando platform is a prototype in FD-SOI 28nm silicon process technology that has recently demonstrated at the ISSCC 17 [1] state-of-the-art power consumption vs. computational power efficiency of 2.9 TOPS/W on realistic CNN configurations.

The Orlando device is a configurable, scalable and design time parametric Convolutional Neural Network Processing Engine. It is powered by

- an energy efficient set of CNN HW convolutional accelerators supporting kernel compression
- an on-chip reconfigurable data transfer fabric to improve data reuse and reduce on-chip and off-chip memory traffic.

The architecture also includes: a power efficient array of DSPs to support complete real-world computer vision applications, an ARM-based host subsystem with peripherals, a range of high-speed I/Os interfacing to imaging and other types of sensors, and a chip-to-chip multilink to pair multiple devices together. The Orlando SoC prototype, shown in Figure 2, integrates an ARM Cortex microcontroller with 128kB of memory, eight DSP clusters (2 DPSs, 4-way 16kB instruction caches, 64 KB local RAMs and a 64kB shared RAMs), and a reconfigurable dataflow accelerator fabric (IPU). Such fabric connects high-speed camera interfaces with sensor processing pipelines, croppers, color converters, feature detectors, video encoders, 8 channel digital microphone interface, streaming DMAs and 8 convolutional accelerators. The chip includes 4 SRAM banks each with 1MB, dedicated bus port, and fine-grained power gating, to sustain the maximum throughput for convolutional stages fitting CDNN topologies reducing the need to access external memory to save power.

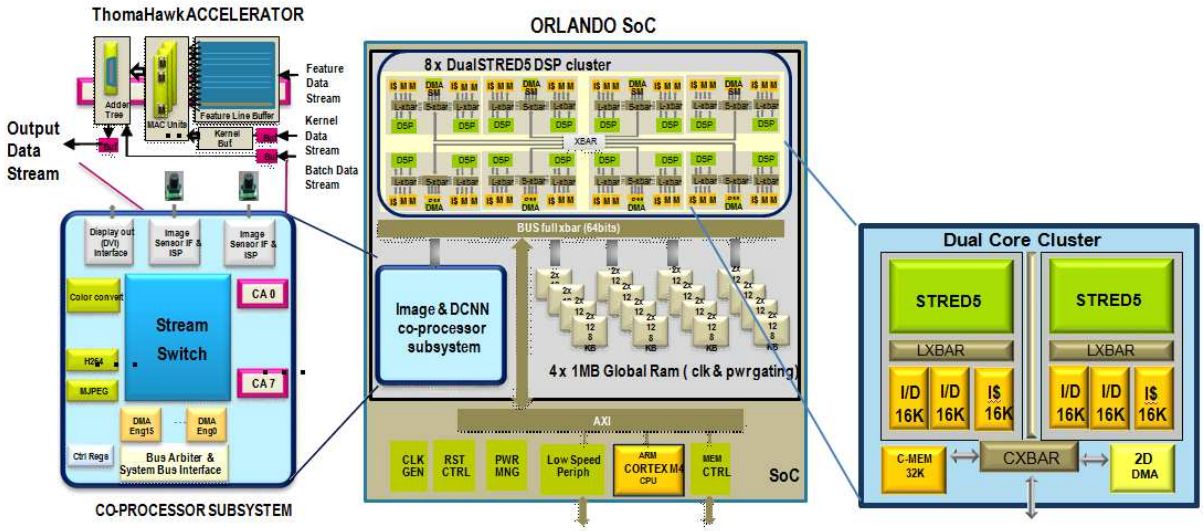

Figure 2: Orlando SoC prototype

The execution of the CDNN layers is covered by the 90% by the hardware convolutional accelerators.

The prototype chip in FD-SOI 28 technology adopts mono-supply SRAM based single well bitcell with low power features and adaptive circuitry to support a wide voltage range from 1.1V to 0.575V, and leverages a GALS clocking architecture to reduce the clock network dynamic power and skew sensitivity due to on-chip variation at lower voltages. A power consumption of 41mW on a typical DCNN algorithm (AlexNet [2]) is achieved with a peak efficiency of 2.9 TOPS/W.

The chip can reach the clock frequency of 1.175 GHz. The peak performance of each of the 16 DSPs is of 4.7 GOPs (dual 16b MAC loops, where a MAC corresponds to two operations, ADD + MUL), and the peak performance of each of the 8 Convolution Accelerators is of 84.5 GOPs. Thus, the chip reaches the total performance of more than 750 GOPs.

In Orlando, as common in low-energy IoT end-nodes, the programmer has access to significant introspection in terms of performance counters, thermal, power and leakage monitors, etc. and to many operating point control knobs (e.g. power modes, voltage/frequency scaling) to be able to adapt to various application requirements and effectively minimize the energy necessary to perform the target algorithm given a (performance, accuracy) target.

3.2.1 Available computing resources

The resources available in Orlando are:

- 16 DSPs (8 clusters of 2 DSPs each)
- 8 Convolutional accelerators (max kernel size 11x11)
- 16 Stream Engines (DMAs)
- 2 Sensor Interfaces
- A H264 Encoder
- A Motion JPEG Encoder
- 4 MB of Global RAM
- 64 KB x 16 of RAM local to the DSPs

Depending on the type and dimensions of the kernel that must be implemented, a HW accelerator or one or more DSPs can be used. Moreover, if the required kernel size or feature size is larger than the one supported

by the HW accelerators (or by the implementation done on the DSPs), multiple iterations can be necessary, saving intermediate results in RAM.

3.2.2 Orlando programming model

Orlando is a research platform and it is not provided with the high-quality programming tools that usually are associated with off-the-shelf SoCs. Nevertheless, a gnu-based C/C++ toolchain exists for compiling, linking and debugging applications for Orlando. An IDE also exists integrating all these tools.

No real OS has been developed for Orlando. A simple system of remote procedure calls (RPC) exists for simplifying the exploitation of the parallelism given by the 16 DSPs. After partitioning the application in functions, one of the cores starts sending requests (dispatching tasks) to the other cores to execute the functions, guaranteeing that precedence rules among functions are respected. This mechanism exploits the fact that all memory can be accessed by all cores. Of course, latency of memory accesses depends on the distance from the core performing the access and the memory block destination of the memory access, so particular attention must be payed to where data must be allocated w.r.t. the various blocks of memory contained in the system.

The hardware accelerators are equipped with control registers which can be used to program them to perform the specific tasks required by the application. A medium level software API exists for simplifying programming of those control registers. A high-level API is being developed for further simplifying the use of the HW blocks. If DSPs will be used for implementing CNN kernels, those software instances of kernels could be managed by this high-level API too.

This high-level API will define a series of functions for instantiating and using kernels of different types. The two phases, instantiation and use, are different because, especially if HW accelerators are used (but the same can be true if DSPs are used), the same kernels, with the same characteristics, will be used for a series of different inputs (e.g. consecutive frames coming from a video sensor); in this case, instantiation can be done only once.

As an example, the following code could be the prototype of a function instantiating a Convolution kernel:

```
bool InstantiateConv(unsigned int id,
                     void *pointer_to_input_tensor,
                     void *pointer_to_output_tensor,
                     void *pointer_to_weights,
                     void *pointer_to_bias,
                     unsigned int stride,
                     unsigned int number_input_features,
                     unsigned int number_output_features,
                     unsigned int feature_height,
                     unsigned int feature_width,
                     unsigned int precision /* 8bit or 16 bit */);
```

This function returns `true` if the instantiation was successful and `false` otherwise. The first parameter of the function is an identifier of the Convolution kernel (e.g. index of the Convolution accelerator that must be used or of the DSP implementation of the Convolution kernel). The meaning of the other parameters is straightforward. A more sophisticated API can be designed where a scheduler can 'allocate' the different instances of the kernels of the same type and return their identifiers (the kernel identifier, from an input parameter, becomes a return value of the API function).

The prototype of a function for executing the Convolution kernel on a set of data could be the following:

```
bool ExecuteConv(unsigned int id);
```

This high-level API will be developed for all the 'typical' kernel types (at least convolution, max pooling, average pooling and ReLU) and will hide the details of their implementation to the programmer (for example, whether they are implemented in HW or as code running on a DSP).

3.3 Off-the-shelf embedded platform

In this section, we will provide an overview of the Snapdragon 845 platform which is the choice of preference to be used as a COTS platform. The architectural block diagram of 845 is shown in Figure 3.

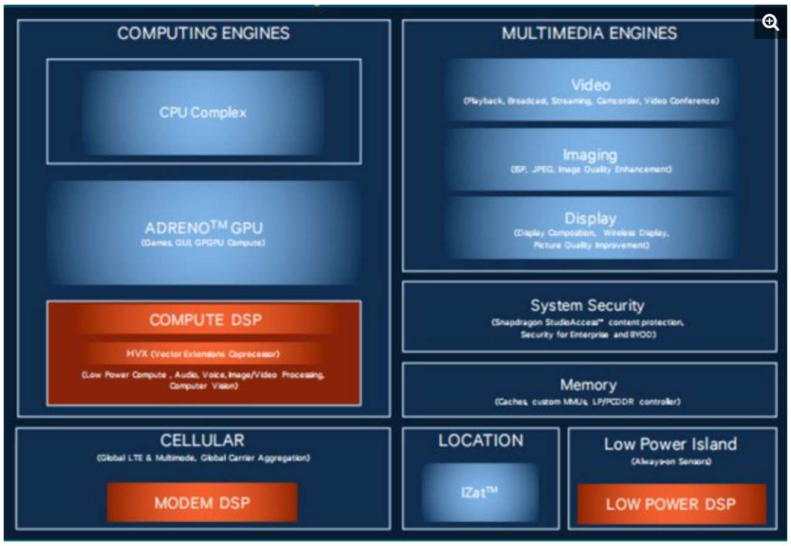

Figure 3: The architecture of the Qualcom SnapDragon 845 SoC

3.3.1 Available computing resources

SnapDragon 845 is the latest SoC released by Qualcomm, which is based on TSMSC 10nm LPP technology. It features a powerful Quad-core processor, a GPU (Adreno 630) and also a very powerful DSP (Hexagon 685).

Central Processing Unit

Integrates 4x Kryo 385 cores (Cortex-A75) at up to 2.8 GHz (max) for performance and 4x Kryo 385 at 1.8 GHz (max) for efficiency, organized in a dual-cluster, heterogeneous configuration. The clusters are optimized to operate at different frequencies and power levels, sort-of like ARM's big.LITTLE approach.

Graphics Processing Unit

The Adreno 530 GPU is a powerful device able to run in variant clocks in the range of 133-710MHz and featuring a nomimal comuting capacity of 727 GFLOPs. One of the major features is its ability to use of data compression when moving data around within the GPU in order to reduce power consumption.

In addition, and in order to facilitate heterogeneous computing, the GPU and CPU can both snoop into each other's cache, enabling better sharing of data, since both processors use 64-bit virtual addresses.

Adreno 530 supports the latest graphics API standards, including OpenGL ES 3.1 + Android Extension Pack, DirectX 12, and Vulkan (once ratified by Khronos). Like the Adreno 430, the 530 includes a dedicated fixed-function block in hardware for accelerating tessellation. Adreno also supports the OpenCL 2.0 GPGPU standard.

Digital Signal Processing Unit

SnapDragon 820 as its predecessors, also features a powerful DSP which is now the Hexagon 685 DSP, which is a specialized type of processor that can do certain tasks many times faster or more efficiently than a CPU. The Hexagon 685 DSP has a highly efficient multi-thread programmable compute engine that is programmed much like a multi-core CPU. It is an advanced, variable instruction length, Very Long Instruction Word (VLIW) processor architecture with hardware multi -threading. It also enables concurrent execution of both audio and imaging tasks, which should increase performance and battery life. Finally, it has been optimally configured to support AI functionality and the Qualcomm SNPE SDK with Caffe, Caffe2 and TensorFlow Support.

3.3.2 Snapdragon programming model

SnapDragon 845 is a very powerful commercial SoC, which as an evolutional development in a series of SoCs, features a very mature programming environment able to efficiently program the computing units individualy, to allow robust integration of software code to a seamless set, and to allow efficient code implementation. To that end, Qualcomm offers, per computing unit, a number of software tools and SDKs, that the programmer can use so to carry out any task.

Central Processing Unit

The CPU as an ARM variant can be programmed with compilers supporting the ARM architecture. However, Qualcomm also offer the SnapDragon LLVM compiler, which incorporates a number of optimizations and customizations, especially accustomed to the Kryo CPU. For debugging, there are also debuggers for the Ecipse and Visual Studio.

Graphics Processing Unit

For the GPU, there is Adreno SDK, and a number of special drivers, which allow user to program the device by using one of the supported standards (OpenCL, Vulcun etc).

Digital Signal Processing Unit

For the DSP there is the Hexagon SDK, a powerful set of tools that enable the optimal programming of the device.

Heterogeneous programming

For the syenergistic or heterogeneous programming of the resoources, Qualcomm offers the Heterogeneous Compute SDK and also the SnapDragon Power Optimization SDK for the optimization of the power consumption of the SoC.

4 References

[1] "A 2.9 TOPS/W Deep Convolutional Neural Network SoC in FD-SOI 28nm for Intelligent Embedded Systems", Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, Paolo Zambotti, Manuj Ayodhyawasi, Harvinder Singh, Nalin Aggarwal, 2017 IEEE International Solid-State Circuits Conference

[2] https://en.wikipedia.org/wiki/AlexNet