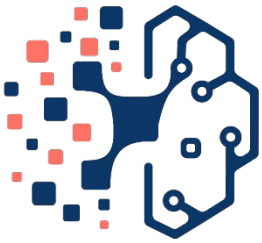

BONSAPPS

AI-as-a Service for the Deep Edge

D2.7 Framework for optimized network tuning and deployment on PULP architectures

Grant Agreement No.	101015848
Project Name	BonsAPPs
Work Package No.	WP2
Lead Beneficiary	UNIBO
Delivery Date	31st December 2021
Author(s)	Simone Benatti (UNIBO)
Contributor(s)	Alessio Burrello(UNIBO),Miguel de Prado(BCA)
Editor(s)	UNIBO, BCA
Reviewer(s)	NVISO, HES-SO
Nature [1]	Report
Dissemination Level	Public

[1] 1 CONFIDENTIALITY LEVEL:
PU = PUBLIC
PP = RESTRICTED TO OTHER PROGRAMME PARTICIPANTS (INCLUDING THE EC SERVICES);
RE = RESTRICTED TO A GROUP SPECIFIED BY THE CONSORTIUM (INCLUDING THE EC SERVICES);
CO = CONFIDENTIAL, ONLY FOR MEMBERS OF THE CONSORTIUM (INCLUDING THE EC SERVICES).
INN - INTERNAL ONLY, ONLY THE MEMBERS OF THE CONSORTIUM (EXCLUDING THE EC SERVICES)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101015848. Neither the European Commission (EC) nor any person acting on behalf of the Commission is responsible for how the following information is used. The views expressed in this document are the sole responsibility of the authors and do not necessarily reflect the views of the EC

Document Revision History

Version	Date	Modification Reason	Modified by
V0.1		Initial version of the deliverable	Simone Benatti (UNIBO)
V0.2		Internal review	Miguel de Prado (Nviso)
V0.3		Internal review	Nuria Pazos (HES-SO)
V1.0		Final version of the deliverable	Nuria Pazos (HES-SO)

Abbreviations

AI: Artificial Intelligence
BMP: Bonseyes AI Marketplace
DNN: Deep Neural Network
NEMO: NEural Minimizer for pytOrch
PULP: Parallel Ultra Low Power
RISC: Reduced Instruction Set Computer

Executive Summary

This document includes the structure and the first content of the framework to design and deploy AI tools on ultra-low power multicore platforms. The target platform is PULP, based on the open-source ISA RISC-V.

Table des matières

List of Figures

List of Tables

1 Introduction

The Bonseyes AI Marketplace is a web platform that connects researchers, developers, and companies to procure, collaboratively build, and trade AI Applications. Its goal is to facilitate collaboration between researchers and industry to speed up the process of building and deploying AI-base solutions to solve real-world challenges defined by the industry.

BonsAPPs will provide to researchers, data scientists, developers and industries the various number of AI Artifacts, e.g., AI papers, datasets, assets, applications and embedded boards. Users can search, browse, and bookmark AI Research from the collection, as well they can create, publish, download, sell and buy AI Artifacts from the AI Marketplace.

The BonsAPPs AI-as-a-Service layer (AIaaS) will provide access to low-end devices and edge platforms , to enable more pervasive AI applications, in the modern IoT scenario. It will be scalable, providing appropriate support to both end-users and AI Talents[1] in a way that does not require high-intensity involvement by technical experts. Regarding edge processors, it is paramount to maximize energy efficiency, coupling the execution of computationally intensive tasks with reduced power envelop. Parallel programming and multicore accelerated architectures are gaining traction in the domain of AI deployment on ioT devices[3]. To help evaluating the deployment of AI frameworks on such microcontroller, the BonsAPPs service will provide a complete end to end framework, which helps AI talents to gather the maximum performance from these architectures.

In detail, this deliverable supports the RiscV-based GAP8 (Figure 1) processor, which is commercially available from Greenwaves technologies [2]. In chapter 2 we describe the PULP architecture, which is the basic structure of the GAP processors. Then we present the quatization tools we intend to use for deploying neural networks on our the GAP8 multicore platform. Finally, we describe the docker, available on the BBonseyes AI Marketplace, to quantize and deploy a network on the HW target.

Figure 1 Gapuino board based on GAP8

[1] AI Talents are researchers, PhDs/post-docs, engineers/developers or data scientists that have capabilities to resolve AI challenges

2 PULP Architecture

This section provides an insight to the PULP multicore architecture. We briefly describe the cluster and the connected peripherals, and we give some pointers for documentation and other resources for a detailed description of the ISA and the architecture.

2.1 Documentation

- https://pulp-platform.org/: Official site of the PULP platform
- https://pulp-platform.org/pulp_training.html: In-depth tutorials to understand and learn the PULP architecture, software developer kit, and tools. Includes video lessons and tutorials on a ready-to-use virtual machine.
- https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html: GreenWaves' GAP8, an open-source, multi-core platform built upon the PULP paradigm.
- https://gvsoc.readthedocs.io/en/latest/: GVSOC, light and flexible instruction set simulator which can simulate GreenWaves' GAP8 IoT application processor. Full applications with real device drivers (cameras, microphones, LCDs) can also be simulated.

2.2 PULP Architecture

The PULP (*Parallel processing Ultra Low Power platform*) project was born to respond to the unmet demand for high-computational power at the edge with a low power budget.Since its creation, many chips have been produced following its philosophy, available at https://pulp-platform.org/implementation.html.All the existing chips share the following features:

- Low-power platform, capable of switching to a sleep state with minimal power consumption when not performing tasks.
- High-performance on demand, to manage high frame-rate requirements with the lowest energy budget
- High flexibility and programmability, to keep on track with the rapid development in the computer vision field of study.

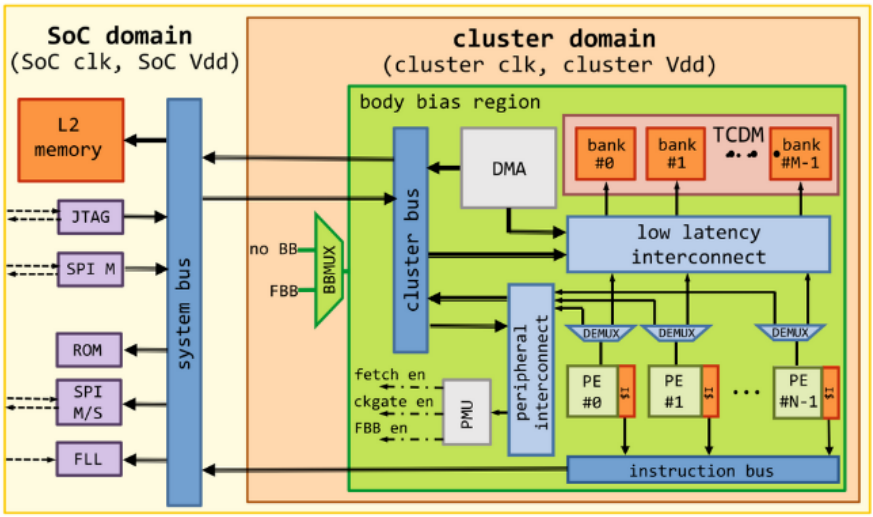

Figure 2 PULP cluster and SOC architecture

In order to achieve high computation performances, PULP uses a cluster of OpenRISC cores, typically 4 to 8. They are called the Processing Elements (PEs) and allow various degrees of data or task-level parallelism. The cores have been optimized to reach high instruction-per-cycle values over a wide range of applications, including control-intensive code. PEs share the same L1 multi-banked tightly coupled data memory (TCDM). The TCDM has several ports equal to the number of PEs, allowing concurrent access to different memory locations. Each PE also has a private instruction cache, but no data cache. TCDM size is variable, but usually smaller than the L2 memory.

A multi-channel DMA allows for fast memory transfers between cores, L2 memory (32 to 128 KB range) and peripherals. It's connected to the TCDM with a low latency interconnect, the same used by the PEs. This eliminates any form of internal buffering when managing L1 data transfers. The cluster domain is connected to all external resources and peripherals via a peripheral interconnect.
In order to provide energy efficiency, each core can operate on private voltage and frequency. To do so, a Frequency-Locked Loop (FLL) is implemented on SoC level. A set of clock dividers (one for the SoC, and one for each cluster core) can divide the FLL-generated clock frequency. For the voltage, a Body Bias Multiplexer (BBMUX) is used. It works synergically with the Power Management Unit (PMU) to quickly switch each part of the architecture between normal and "boost" mode, whenever the computed task needs it. The PMU guarantees that the different operating modes stay transparent to the software, by generating control signals for fetch enables, clock gating units, and the BBMUX.

Other peripherals integrated are a set of two Serial Peripheral Interfaces (SPIs), one for master and one for slave, a bootup ROM, and a JTAG interface used for testing purposes. The SPIs can be set in single or quad mode depending on bandwidth, can be linked to various off-chips components (like sensors), up to 4 slave peripherals. The peripheral architecture allows the system to be in two different operating modes:

- Slave mode: PULP acts as a multi-core accelerator of a standard host processor. The host must load the application on PULP L2 by using the SPI master interface and synchronize the computation with dedicated signals.
- Standalone mode: PULP detects external flash memory on the SPI master interface. If none are linked L2 is used instead.

3 Quantization and deployment tools

Research in the domain of Artificial Intelligence represents scattered resources across the internet, making searching, finding, and following specific research artifacts hard. In order to mitigate that problem, Bonseyes AI Marketplace acts as an aggregator platform, allowing one with an interest in it to find, browse and follow latest research in the AI domain. Research resources on the Marketplace are split into two categories:

- Research that represents papers and associated implementations of the papers,
- Datasets that provide references to the publicly available collections of domain data.

From developers' and data scientist perspective, they represent a starting point that can help in the process of AI systems development. Indeed, BonsAPPs project aims to provide quantization tools to ease the optimized deployment of AI frameworks, enabling their pervasiveness on the market. In this chapter, we introduce NEMO, a quantization tool, and DORY, a deployment tool specially tailored for PULP-based architectures.

3.1 NEMO

Quantization is a critical point to enable low-energy execution of neural networks on edge-constrained devices. Recent developments show that applying 8-bits integer quantization reduces the memory occupation of 4X and the energy per inference by an even higher factor, without lose in accuracy.

To deploy network on PULP devices, we decide to apply post-training quantization or quantization-aware training. The first uses a simple tuning dataset to convert a floating point network with frozen weights to its integer counterpart. The data are used to set the minimum and maximum margins of the intermediate activations of the network. The latter exploits the whole training set to re-train the network and produce the so-called fake-quantized network, a network whose weights are yet floating-point, but discretized to 2^N values. In both the cases, the final step consists of translating the network in a full-integer one, where the values of the integer values are associated to their floating point counterpart. Note that on the deployment platform, the whole execution is performed with only integer values and integer arithmetic.

We perform all these steps with our tool, NEMO, Neural Minimizer for pytOrch, which allows choosing between post-training and quantization aware training and performing the graph transformation from floating point to integer with few lines of python code. This tool is fully integrated in the container that we provide for the deployment of neural network on the edge and it produces outputs as open neural network exchange (ONNX) graphs, where both the topology and the parameters of the target network are stored.

The quantization technique applied is the linear quantization to both activations and weights. Noteworthy, using non-linear quantization would lead to a slightly higher accuracy, at the cost of a much more complicated execution on the target platform.

In the docker file a scrip has been included to run the MobilenetV1 example:

```
python3 test.py

source sourceme.sh

-number of board to use, 10 for gapuino-

cd dory/dory_examples/

python3 network_generate.py --sdk=gap_sdk --network_dir=../../

cd application/

make clean all run platform=gvsoc CORE=8
```

To flash the example directly on the board the parameter

```
platform=board
```

is required at compiling time, and the USB driver must be exposed while launching the docker

```
--privileged -v /dev/bus/usb:/dev/bus/usb
```

3.2 DORY

The second tool employed for the deployment of neural networks on PULP nodes is DORY, our memory-oriented deployment tool . This tool fills the gap between the full integer graph and the final C code needed to run the network on the target. Specifically, the tool solves the following problems:

- Starting from the .ONNX graph produced by NEMO, it fuses operators to match the ones present in the available kernels for PULP platforms (***ONNX Decoder***).
- For each layer, it analyses the memory requirements and creates the corresponding mapping on the different level of memories to minimize the latency to execute the layer (**Layer Analyzer**);
- It parses the transformed graph to generate the execution network file (**Network Parser**).

DORY targets a compute node with three levels (L3, L2,and L1) in the memory hierarchy. It supports L3-L2 and L2-L1 tiling of both weights and activations. Storage of weights in L3 (> 512 kB) is essential for the deployment of most non-trivial networks. On the other hand, activations' tiling is typically necessary only for networks working on high-resolution images with big spatial dimensions, which are rare in the edgecomputing domain.

The final output of DORY is an ANSI C file that embodies the whole DNN execution and can be compiled for the target platform.

3.2.1 ONNX Decoder

The first operation performed by DORY is decoding the input ONNX graph representing an already quantized DNN, and reorganizing it in a set of layers. In DORY, a *layer* corresponds to a canonical sequence of operations performed by distinct ONNX graph nodes (Figure 4). Each layer includes:

i) Linear/add/pooling operation,
ii) an optional Batch- Normalization operation,
iii) a Quantization/Activation operation.

Each DORY layer uses quantized inputs, outputs, and weights, while the representation of any temporary datais 32-bit signed integer.

3.2.2 Layer Analyzer

In the first optimization phase, DORY layers are considered separately from each other, using only weight dimension information from the previous layer. The layer analyzer includes two submodules: a platform-agnostic *tiling solver*, and a *SW-cache generator*.

3.2.2.1 DORY Tiling Solver

In the following discussion, we denote a buffer residing in Li memory as Li_t, where **t** is the name of the tensor. The Solver relies ona 2-step engine, which solves the L3-L2 tiling constrained problem first, and the L2-L1 one afterwards. With L3-L2 tiling, we enable storing activations and weights in the L3 off-chip memory instead of the on-chip L2. With respect to tools that do not support L3 tiling for activations, such as Tensorflow Lite Micro, this feature enables the support of significantly larger layers. The Solver verifies whether the layer memory occupation fits the L2 memory input constraint orneeds to be stored in L3:

$$L2_{w,next} + L2_{w,curr} + L2_x + L2_y < L2$$

We search an L3 tiling solution using a five-stage cascaded procedure. At each stage, we try to tile a different selection of buffers to fit the constraint of above equation. Whenever possible, the tiler tries to avoid L3-L2 tiling of output activations, which always requires a double number of transfers (from L2 to L3 when produced, and from L3 to L2 when consumed by another layer). Instead, the tiler tries to keep output activations in L2 as much as possible. If a stage satisfies the equation, the L3-L2 Tiling Solver is stopped and the dimensions of tiles are saved. Otherwise, the next stage is tried.

I. *stage 0.* L3-tile x, w, y = OFF, OFF, OFF. If the equation is directly satisfied, we proceed without L3-L2 tiling.

II. *stage 1.* L3-tile x = ON. This solution is selected when the output of the previous layer was tiled in L3, and therefore input tiling cannot be avoided. Tiling is performed along the h_x (height) dimension of the input, to avoid 2D transfers at the L3-L2 interface. The tiler splits the layer in a series of identical ones that work on a different stripe of the input image.

III. *stage 2.* L3-tile w = ON. Weight tiling is enabled on the C_y (channels) dimension, dividing the layer in a set of smaller layers that work on different channels of the output image with $C_y^1 < C_y$. This solution can only be selected when the output of the previous layer is already in L2.

IV. *stage 3.* L3-tile w, y = OFF, ON. Weight tiling is disabled while output tiling is enabled: the approach is similar to input tiling, but requires doubling the DMA transfers for the tiled tensor across the full network execution.

V. *stage 4.* L3-tile w, y = ON, ON. The L3 tiling is enabled on both buffers, *y*, *weights*. This solution is selected when no other solution can fit L2.

After the L3 tiling step, the DORY solver processes the layer to find a suitable L2-L1 tiling scheme, which requires more effort due to the typically small sizes of L1 memories. Compared to high-end computation engines, with much larger memories, a suboptimal sizing of the tensors for the L1 small MCUs memory can be even more detrimental in terms of performance. DORY abstracts this as a Constraint Programming (CP) problem and exploits the CP solver from the open-source OR-Tools developed by Google AI to meet hardware and geometrical constraint (e.g., C^t for output and weights must be the same), while maximizing an objective function, e.g. maximizing the utilization of the available L1 memory.

Topological and geometrical constraints are inserted to respect the relationships between each tensor's characteristic dimensions and other parameters of a layer.

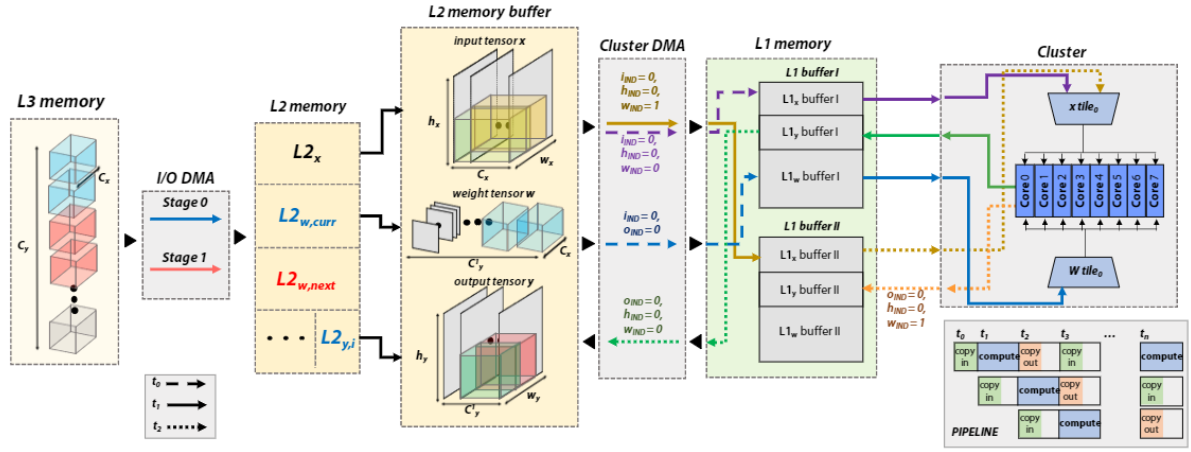

Figure 3 DORY L3-L2-L1 layer routine example. On the left, the I/O DMA copies weights tile in case only Cy is L3-tiled. Two different buffers are used for L2w. Then, the Cluster DMA manages L2-L1 communication using double-buffering, while the cores compute a kernel on the current tile stored in one of the L1 buffers

3.2.2.2 DORY SW-cache Generator

The SW-cache Generator is charged of automatically generating C code orchestrating the execution of a whole layer given the tiling solution found by the Tiling Solver. It instantiates asynchronous data transfers and calls to the backend kernels, without any manual effort. DORY uses a *triple-buffering* approach for the communication between L3-L2 and L2-L1 memories: specifically, double-buffering isapplied simultaneously between L3-L2 and L2-L1 (Figure 3), and all data transfers are pipelined and asynchronous. With this approach, we can almost completely hide the memory transfer overhead. While the codegenerator is necessarily not platform-agnostic, the approach we follow can be easily generalized to any computing node with a three-level memory hierarchy.

```
1  LTO: for (o = 0; o < C_y^t; o++)
2    LTH: for (h = 0; h < h_y^t; h  ++)
3      LTW: for (w  = 0; w < w_y^t; w ++)
4        LTI: for (i = 0; i < C_x^t; i ++)
5          dma_wait(L1_{x,load}); swap(L1_{x,load}, L1_{x,exec})
6          dma_async(L1_{x,load} <- L2_x[i, w, h])
7          dma_wait(L1_{w,load}); swap(L1_{w,load}, L1_{w,exec})
8          dma_async(L1_{w,load} <- L2_w[i, o])
9          if (o + h + w + i > 0)
10           DNN_kernel (L1_{x,exec}, L1_{w,exec}, L1_{y,exec})
11 # from 3° iteration: fully operating pipeline
12         if (o + h + w + i > 1)
13           dma_wait(L1_{y,load})
14           dma_async(L1_{y,load} -> L2_y[ o, w, h])
15           swap(L1_{y,load} , L1_{y,exec})
```

Table 1 DORY L2-L1 loop nest implementing the double buffering scheme. At each most internal loop iteration, two asynchronous Cluster DMA calls are made to copy the weights and input activation of the next tile into L1 memory, the basic kernel is executed on the current tile, and one other cluster DMA transfer is executed to copy the output back on the L2 memory

Table 1 provides DORY's scheduling scheme of L2-L1 layer execution, through LTO, LTW, LTH, and LTI loops on output channels, height, width, input channels tiles, respectively. Loop iteration

limits are statically resolved by the *DORY tiling Solver*. Moreover, DORY autonomously controls the complete execution of the layer, by managing padding, stride, and overlap for every single tile (e.g., padding >0 for border tiles whereas padding = 0 for internal ones, when the input padding parameter is > 0). Using statically resolved parameters, we maximize the usage of immediates,reducing load/store operations inside the inner loops of thelayer tiling.

The layer-wise loop nest detailed in Figure 3 is executed in three concurrent pipeline stages:

i) a new computation starts and fill the output buffer that was not used in the previous cycle;
ii) the results of the last cycle are stored back in L2;
iii) a new set of inputs is loadedin L1. At each pipeline cycle, we swap the load and the execution buffer (*swap* operation of Listing 1) to enable double buffering.

3.2.3 Network Parser

```
1  udma_async(L2_{w,load} <- L3_w[I_0])
2  udma_wait(L2_{w,load});
3  LTL: for (i = 0; i < nlayers; i ++)
4  # number of CNN layers
5    udma_wait(L2_{w,load}); swap(L2_{w,load}, L2_{w,exec})
6    if  (layer{i+1} fit L2 && is Conv)
7        udma_async(L2_{w,load} <- L3_w[I_i])
8    Layer{i} (L2_x, [L2_{x2}], [L3_w[I_i]], [L2_{w,exec}], L2_y)
9    # [] optional arguments
10       swap(L2_y, L2_x)
11   if  (layer{i} has residual) # bypass management
12       store  (L2_y->L2_{x2})
13   if  (layer{i} is Sum)
14       delete  (L2_x2)
15       Stack_dealloc(L2_y) # stack control
16       Stack_alloc(L2_x[I_{i+1}])
```

Table 2 DORY network execution loop

After layer-wise tiling has been completed by the Layer Analyzer, DORY uses the information extracted from all the layers to build a network graph, considering every single layer as a callable function. Table 2 showcases theexecution loop of the DNN execution as created by our framework. At each step, three main tasks are concatenated:

i) we transfer from L3 the weights of the following layer.
ii) a new layer is executed pointing to the correct buffers inside the memory stack;
iii) input and output buffer offsets are updated.

Similarly to single layers, the network-wise code is generated automatically without programmer intervention.DORY produces a single function that can be called insidea custom application by passing two externally allocated memory buffers (for L1 and L2) and their maximum size as parameters.

3.2.3.1 Buffer allocation stack & Residual connections

To allocate layer-wise input and output buffers in the L2 memory, we created a two-stack strategy, employing a strategy based on a single bidirectional stack designed to avoid memory fragmentation and enable the execution of a sequence of differently sized layers. Buffers are allocated/deallocated from the bufferallocation stack, which is constituted by two concurrent Last-In-First-Out stacks growing in opposite directions. At the end of each layer's weight buffer allocation, we reverse the end of the stack for the next memory allocations. By construction, the bidirectional stack is at worst as big as two concurrent stacks growing in the same direction. For example, in a simple case without residual connections the dimension of our *bidirectional stack* is:

$$D_{stack} = \max_i(L2_{x,i} + L2_{w,i} + L2_{w,i+1} + L2_{x,i+1})$$

which is always less or equal than the size of two concurrentstacks $D_{stack,1}$, $D_{stack,2}$ due to the triangle inequality.

Before executing the *i*-th layer, the allocator manages the weight buffer $L2_{w,i}$ and output buffer $L2_{y,I}$; notice that $L2_{x,i}$ is already allocated as the $L2_{y,j}$ of a previously executed *j*-th layer (or the input of the network). To manage residual connections, each $L2_{y,i}$ buffer has a lifetime counter associated. To allocate a buffer in the stack for the *i*-th layer, as shown in Figure 3 :

1. one of the two corners of the stack is selected depending on a begin_end flag that is switched at each new weight allocation;
2. the allocator deallocates the last $L2_{w,\,i-2}$ buffer on the corner;
3. the allocator checks if $L2_{y,\,i-2}$ has its lifetime counter set to 0; if so, it is deallocated;
4. $L2_{y,\,i}$, $L2_{w,\,i}$ are allocated in order in the selected corner (with $L2_{w,i}$ nearest to the pointer);
5. the lifetime counter of $L2_{y,\,i}$ is set to the lifetime of the activation buffer, i.e., the number of layers to be executedbefore its deallocation.
6. all lifetimecounters are decreased by 1.

The buffer allocation stack is naturally suited to execute a network with different branches (i.e., residual connections). DORY always prioritizes the branch with the highest number of nodes. The overall size of the stack is computed offline statically, taking into account all residual connections: its dimension depends on the maximum sum of memory of two subsequent layers plus all the residuals from the previous layers.

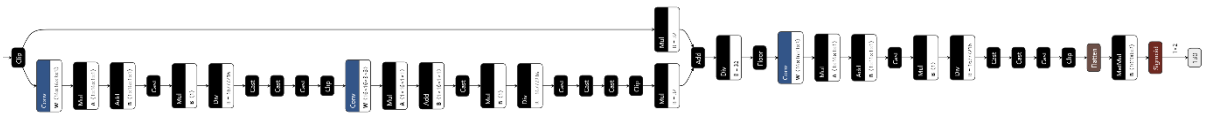

Figure 4 Network structure of ONNX

4 Docker structure

In this section, we describe the Docker file structure that can be used to quantize and deploy neural networks on the GAP processor. The created docker imagine allows the setup of the PULP-based platforms, quantization and cross-compilation of the selected AI-App and the management of the platform. Once the repositories are pushed to the BonsAPPs' gitlab account, the docker images will be built and stored in the BMP.

4.1 Download and installation

- Download and install docker on your machine from https://gitlab.com/bonseyes/platforms/mcu-workflows/gap8
- (For Windows users only) install PowerShell from here.
- Start the Docker daemon. On Windows, this is done by opening Docker Desktop.
- Open a shell prompt and run this command line:
  docker pull dequi/gap-sdk
- Once the download finishes, you can run the Docker image into a container with the following line:
  docker run -it dequi/gap-sdk:latest
- That's it! You are now ready to work with the GAP_SDK, NEMO and DORY in a command-line based environment.

```
PS C:\Users\alber> docker run -it dequi/gap-sdk:latest
root@c540c21553c5:/gap_riscv_toolchain_ubuntu_18/gap_sdk#
```

4.2 Docker image description

The docker image you can download following the guide in the paragraph above (4.1) is based on the Ubuntu 18.04 LTS "Bionic Beaver" release. It is a fully working Linux machine already set up for developing and implementing AI-Apps on PULP-based architectures (section 2) using the GAP SDK, and the tools explained in section 3 of this document, NEMO and DORY.

There is no need to install anything on the container, as the dependencies required are all already satisfied. As the container is based on an Ubuntu image, to date the only user interface available is through command lines. Please refer to the official Docker documentation if you want to mount your volumes in the container's workspace here.

4.3 GAP_SDK

Once the container is launched, the default working directory the container is */gap_riscv_toolchain_ubuntu_18/gap_sdk/*, where all the necessary tools for deploying your applications are stored.

The content of this directory is as follows:

- **Docs**: Runtime API, auto-tiler, and example application documentation.
- **rtos**: Directory with the available runtime operating systems available, including FreeRTOS (https://www.freertos.org/features.html) and PULP-OS, a simple open-source operating system developed by the PULP project. Both use the open-source PMSIS as system layer to provide common APIs for applications.
- **sourceme.sh**: A script for configuring the GAP SDK environment.
- **Examples**: Examples of runtime API usage.
- **nemo**: Directory containing the installation of the NEMO tool for minimizing neural network models developed in Pytorch.
- **dory**: Directory containing the installation of the DORY tool for deploying DNNs on MCU.

More information about the *GAP_SDK*, including how to compile and run the "hello world" example, allocate memory, use event and thread scheduler APIs, use the DMA, synchronize cluster cores, measure performance and using peripherals, can be found at https://greenwaves-technologies.com/manuals/BUILD/PULP-OS/html/index.html.

4.4 GVSOC

A lightweight and flexible instruction set simulator which can simulate GreenWaves' GAP8 IoT application processor is already installed in the docker image. You can run your applications on this simulator by using the ***platform=gvsoc*** option when compiling and running with the gap_sdk toolchain. The virtual platform is simulating the architecture, which is described by the specified system configuration, described with a JSON file.

This can be first done through the option *--config-file* to give the path of the JSON file. This can be either an absolute path or a relative path, in which case the config isearch in the paths given by the environment variable SDK_CONFIGS_PATH, which contains a list of possible paths separated by ":".
The configuration is a high-level description of the architecture, where all important properties are specified (e.g., memory sizes). This high-level view of the architecture is used to generate a low-level and detailed view of the architecture which is used by gvsoc to know what to instantiate, configure and connect. Both levels can be customized by the user. The high-level view is called the template, and can be customized to easily change architecture properties such as memory sizes. The low-level view is called the configuration and can be customized to change properties of one specific component, such as a specific behaviour of one core.

Options to the virtual platform are passed by customizing the system configuration.
This can be first done using the option *--property=<path>=<value>* to specify a property in the JSON file to be overwritten; <path> is giving the property path in the JSON file where the property must be overwritten; and <value> the value to be set. As a JSON file is hierarchical, <path> describes a hierarchical path, like a file system path. As described in the previous section, a property can be changed either in the template or in the configuration. Any property beginning with *config/* will change a property in the configuration while the others will change it in the template.

The virtual platform allows dumping architecture events to help developers debugging their applications by better showing what is happening in the system. For example, it can show instructions being executed, DMA transfers, events generated, memory accesses and so on.
This feature can be enabled and configured through the option *--trace*. This option takes an argument which specifies a regular expression of the path in the architecture where the traces must be enabled, and optionally a file where the traces should be dumped. All components whose path matches the specified one will dump traces. Several paths can be specified by using the option for several times.

One difficulty is usually to find out which paths should be activated to get the needed information. One method is to dump all the events with *--trace=.**, then find out which one are interesting and then put them on the command line. Here are the paths for the main components (note that this can differ from one chip to another):

- **/sys/board/chip/cluster/pe0:** Processing element, useful to see the IOs made by the core, and the instruction it executes. You can add /iss to just get instruction events.
- **/sys/board/chip/cluster/event_unit:** Hardware synchronizer events, useful for debugging inter-core synchronization mechanisms
- **/sys/board/chip/cluster/pcache:** Shared program cache accesses
- **/sys/board/chip/cluster/l1_ico:** Shared L1 interconnect
- **/sys/board/chip/cluster/l1/bankX:** L1 memory banks (the X should be replaced by the bank number)
- **/sys/board/chip/soc/l2:** L2 memory accesses
- **/sys/board/chip/cluster/dma:** DMA events

The virtual platform is by default simulating only a stand-alone chip with a few default devices which are required to boot a simple example. Device models such as camera, flash or microphones can be connected in order to run full applications.

4.5 DORY Examples

A set of models are already available for testing and learning how to use DORY. The set contains a *MobilenetV1-128* and 4 other custom networks.

You can execute the tool by running the *network_generate.py* python script. This script has different parameters that you can specify, you can see a description of them by running the script with the *-h* option.Some of the most important ones are:

- **--network_dir:** directory of the onnx file of the network.
- **--l1_buffer_size:** L1 buffer size. Does not include stack size.
- **--l2_buffer_size:** L2 buffer size. Does not include stack size.
- **--master_stack:** Cluster Core 0 Stack.
- **--slave_stack:** Cluster Cores 1 to 7 Stack.
- **--sdk:** Selects the SDK to use.
- **--fc_frequency:** Frequency of the fabric controller.
- **--cl_frequency:** Frequency of the cluster cores.

To build and run a new network from scratch, you first must source in the gap_sdk by running in the *gap_sdk* directory (the starting default work directory):

```
source sourceme.sh
```

Select the chip type, GAPUINO_V3 is a good candidate for testing out.
Next, run the following commands in the *dory/dory_examples* directory:

```
python3 network_generate.py –sdk=gap_sdk
cd application
make clean all run CORE=8 platform=gvsoc
```

To generate the MobileNet V1-128 quantized in 8 bits, ready to be deployed.
The application will also run a simple check on the network deployed, showing the number of MACs and Cycles required, and checking the correctness of its results.

5 Conclusions

This document has introduced the complete toolchain for deploying deep neural networks on PULP based platforms as GAP8 processor.

The document is intended to show the structure and ground of the docker file that will be used to support developer community on the deployment and implementation of AI framework on PULP processors.

References

1. Prado, M.D., Su, J., Saeed, R., Keller, L., Vallez, N., Anderson, A., Gregg, D., Benini, L., Llewellynn, T., Ouerhani, N., Dahyot, R. and Pazos N., 2020. Bonseyes AI Pipeline—Bringing AI to You: End-to-end integration of data, algorithms, and deployment tools. *ACM Transactions on Internet of Things*, *1*(4), pp.1-25.
2. https://greenwaves-technologies.com/
3. Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, Luca Benini, GAP-8: A RISC-V SoC for AI at the Edge of the IoT, *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*

Bibliography

1. Llewellynn, T., Fernández-Carrobles, M.M., Deniz, O., Fricker, S., Storkey, A., Pazos, N., Velikic, G., Leufgen, K., Dahyot, R., Koller, S. and Goumas, G., 2017, May. BONSEYES: platform for open development of systems of artificial intelligence. In Proceedings of the computing frontiers conference (pp. 299-304).