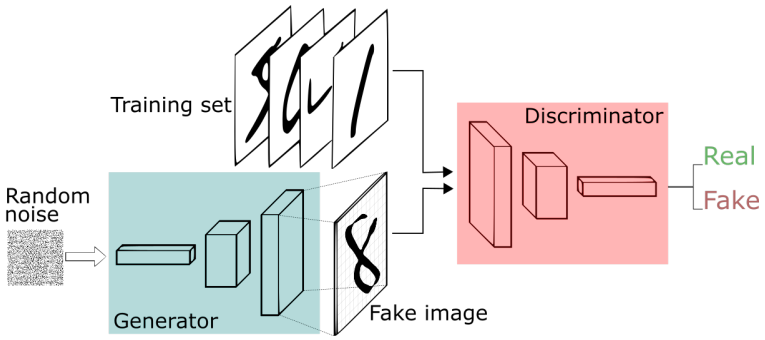
Generative Adversarial Neural Networks on FPGA

Embedded Deep Learning

Alessandro Mileto

March 03, 2023



Credits: https://sthalles.github.io/intro-to-gans/

POLITECNICO MILANO 1863
NECST laboratory

Abstract

As the title suggests, the present project deals with Embedded Deep Learning, id est, the practice of running deep learning algorithms on embedded devices. This work focuses on the acceleration of a neural network over FPGA using the Vitis AI/PYNQ development flow. In particular, a Deep Generative model has been trained, quantized and compiled to run over a Ultra96v2 board equipped with a Deep Learning Processing Unit hardware accelerator, an IP by Xilinx.

1 Introduction

Generative Adversarial Neural Networks are a particular family of Generative Machine Learning models aimed at generating data which are similar to the ones belonging to the training dataset in terms of features distribution and statistical properties. Goodfellow at al. first introduced this family of models in 2016 in [**1**] and since then they gained a lot of momentum. They were also called "The Holy Grail of image generation". Even though they were initially born to support the image generation use case, their adoption is now considerably high in many fields and their long lasting contributions are valuable even when dealing with state-of-the-art generative models. As virtually every deep learning algorithm, GANs benefit from hardware acceleration as an enabler of real world use cases. Xilinx FPGA may be a valid candidate for such task.

2 GANs: a sneak peek

GANs are based on an intrinsically elegant mathematical framework: they consist of a pair of deep neural networks (Discriminator and Generator) playing a **minimax game**, which corresponds to the training phase [**1**]. All along this latter, the Generator tries to fool the Discriminator generating from statistical noise images which are supposed to look real, as if they were drawn from the dataset at hand. In turn, the Discriminator has to distinguish between real and bogus generated samples. As training goes on, the Generator progressively improves its generation capabilities and it becomes harder and harder for the Discriminator to distinguish between real and artificial samples. The game stops as soon as the Discriminator network is no longer able to perform such task. In the end, this latter is dropped and the Generator is kept as a hopefully powerful algorithm for producing from noise (gaussian noise, for instance) samples which are close enough to the ones belonging to

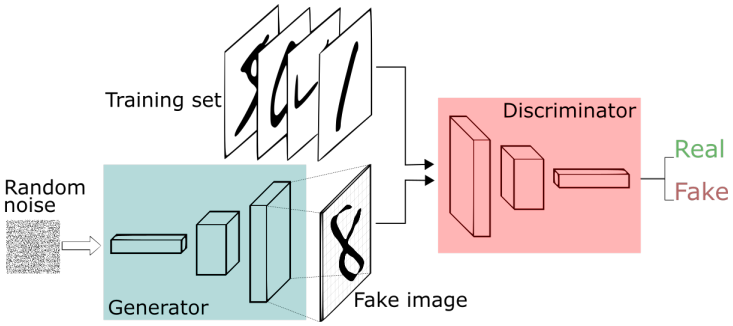


Figure 1: High level view of GANs.

the initially available dataset [**1**].
As stated above, there are many GANs use cases. They might be adopted to create images of people who do not exist, to extend imbalanced dataset through ad hoc data augmentation, to produce counterfeit audio samples for fooling voice recognition systems, to produce image templates to be used by artists as a source of inspiration, to solve difficult planning problems [**2**] and so on.

The following equation shows the loss function to be optimized when performing GANs training.

$$\mathcal{L}(G, D) = E\mathbf{x} \sim p_{\text{data}}(\mathbf{x})[\log D(\mathbf{x})] \qquad + E\mathbf{z} \sim p\mathbf{z}(\mathbf{z})[\log(1 - D(G(\mathbf{z})))]$$

This loss function is a variant of categorical cross entropy, which is used for modelling statistical learning classification problems. The Discriminator is trained in such a way that it tries to minimize it. Conversely, the Generator tries to maximize the loss of the Discriminator.
The networks structure from an high level point of view is represented in 1. Again, The full architecture is made of up two networks. One possible choice for the composition of such networks consists in adopting mainly Convolutional Layers. There are many alternatives but the investigation of the capabilities of such models is out of scope.

3 Deep Learning on FPGA: what, why and how

Deep Learning based applications require considerable computational power and ad hoc hardware acceleration (for example, using GPUs) in order to be appealing for every day use cases. Embedded systems, on the contrary, are

typically constrained in size and power. Such limitations stem from the fact that they are not born with the idea of directly running memory and computationally intensive workloads. Nevertheless, running machine learning algorithms over platforms with a reduced size and price has become pivotal for modern applications [**3**]. Deploying AI over such devices, which are widely spread at the edge of the Internet and are heavily used in ad hoc mesh networks, enables real-time capabilities of consumer applications and provides privacy preservation [**3**]. Moreover, one thing which should be always kept in mind when using embedded devices for deep learning is that many of them are endowed with **programmable logic**: this enables hardware and software designers to create custom hardware platforms capable of managing in a personalizable fashion deep learning workloads. As pointed out before, hardware acceleration is pivotal; besides that, power consumption has to be kept an eye on when dealing with the computing platforms above mentioned. FPGA look like a perfect fit for doing so.
However, as always, there is **no free lunch**: modern applications leverage machine learning models whose computational requirements are still prohibitive for most embedded platforms. For instance, since this work focuses on Neural Networks, there are architectures whose size can easily reach some GB and fitting such a model on small RAM memory banks is still a serious (physical) problem. This implies that deep learning models have to be processed in such a way that they can actually "fit" on the machine at hand. Something has to be sacrificed to adapt an algorithm to the platform used and that "something", most of the times, is its complexity (and accuracy). A trade-off exists: the higher the complexity of the model (usually measured in terms of the number of parameters), the higher its power (and, potentially, its accuracy). In order to properly deploy deep learning model at the edge, several techniques can be used to lower model complexity in an accuracy-friendly way.
However, before pointing them out, it is needed to comprehend how a deep learning algorithm can be modelled by a computational framework. Neural networks were born with the idea of mimicking what happens in the brain. This very idea, however, does not provide us with practical advice on how to put them on a computer. An easier (but yet very flexible) structure to think about, instead, is a graph. In model frameworks, models are mapped over a graph whose nodes represent operations and whose edges are multidimensional arrays named Tensors. [**4**]
Mathematical operations involved range from simple products to convolutions and nonlinear operations such as hyperbolic tangent and so on. Some operators have their own associated parameters used to weigh the inputs. These weights are tuned by means of an ad hoc procedure which ultimately

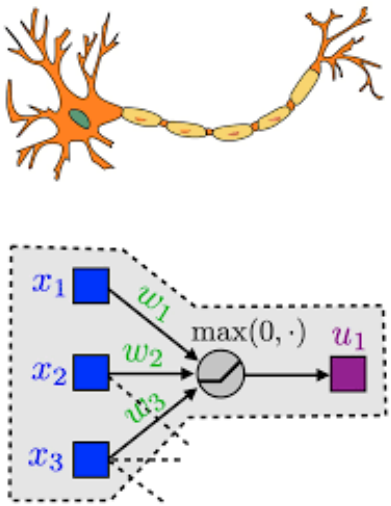


Figure 2: Example of operation carried out by a neural network
Credits: https://mathematical-tours.github.io/book-basics-sources/neural-networks-en/NeuralNetworksEN.pdf.

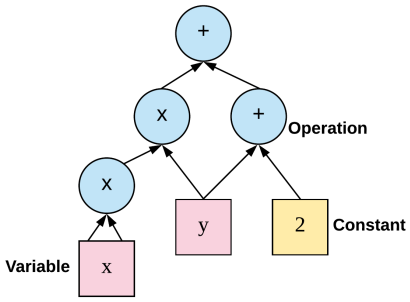


Figure 3: Example of computation graph (Tensorflow)
Credits: https://www.easy-tensorflow.com/tf-tutorials/basics/graph-and-session.

corresponds to an optimization process carried out during what we call the Training phase. Jumping into the bit realm, weights are nothing more than floating point numbers most of the times. Floating point precision may vary across applications and even the same network may be implemented in many ways potentially leveraging different precision values (e.g. FP16, FP32). Nevertheless, moving to integer (lower) precision is often enough for most modern applications whose throughput and memory bandwidth requirements are also stringent.
Mapping floating point arithmetic on **integer arithmetic** improves memory management and eases computation over embedded platforms, lowering power consumption. This former mapping is realized during the process of **Quantization**: floating point numbers are turned into integers trying to preserve as much accuracy as possible. There are several options when dealing with quantization. The two which were investigated during the realization of this work are:

1. quantization-aware training: the model is trained using a reduced fixed precision from the very beginning.

2. post-training quantization (main focus of this work): applicable to whatever floating point model, but potentially less effective than the former since the model has been trained on floating point precision.

Going on, another mechanism to lower the memory footprint of the model craving for efficiency is **pruning**. Model pruning is about reducing model complexity by means of accuracy preserving parameters cutting algorithms. The core theoretical foundation of this that not every parameter is needed when doing inference on an already trained neural network. Some parameters are just unused when performing inference, id est, their presence is not very relevant for the predictions of the model; including them causes overall inefficiency [**3**]. Nevertheless, pruning techniques are quite involved. Sometimes, they even produce model whose structure may not be particularly good for acceleration; moreover, good pruning tools are often distributed under a paid subscription or have to be bought by the relative software house (e.g. Xilinx) and so this technique was not investigated in the current work.
There are several other algorithms which concur in enabling embedded deep learning but, given the very low complexity of the model used in the present work they were not investigated as well.

Features

- Xilinx Zynq UltraScale+ MPSoC ZU3EG A484
- Micron 2 GB (512M x32) LPDDR4 Memory
- Delkin 16 GB microSD card + adapter
- PetaLinux environment available for download
- Microchip Wi-Fi / Bluetooth
- Mini DisplayPort (MiniDP or mDP)
- 1x USB 3.0 Type Micro-B upstream port
- 2x USB 3.0, 1x USB 2.0 Type A downstream ports
- 40-pin 96Boards Low-speed expansion header
- 60-pin 96Boards High-speed expansion header
- 85mm x 54mm form factor
- Linaro 96Boards Consumer Edition compatible

Figure 4: Technical specifications of Ultra96v2.
Credits:https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/

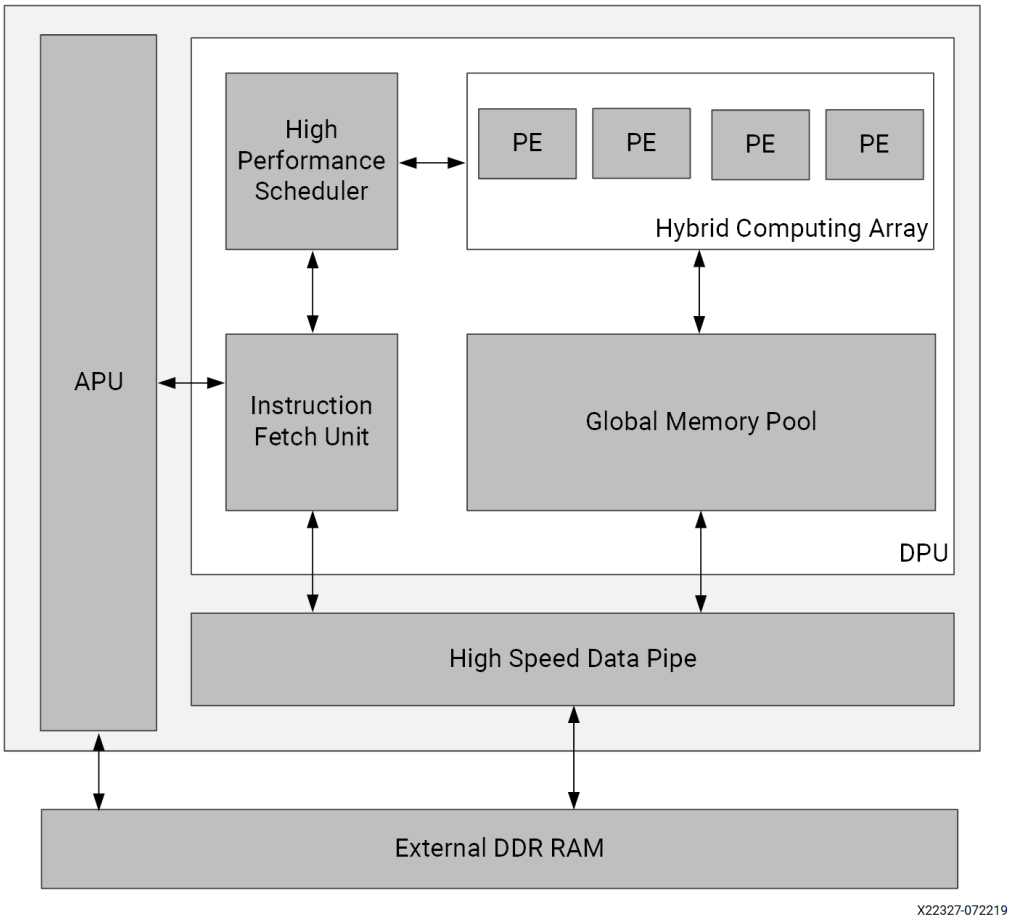


Figure 5: Top-level block diagram of DPUCZDX8G.
Credits:https://docs.xilinx.com/r/en-US/pg338-dpu/Core-Overview/

4 GANs on FPGA: a tale of two worlds

The main focus of this work is the acceleration of Generative Adversarial Neural Networks over FPGA leveraging high level primitives to manage the hardware. In particular, the project revolves around the idea of accelerating the generation of clothing images by means of a Generator network (one half of a GAN) using an intellectual property by Xilinx devised to speed up graph computations. This IP is named DPU (Deep Learning Processing Unit) and it is a custom soft core realized with tensor operations in mind. The ISA of the DPU comes with different implementations whose complexity and capabilities vary across the Xilinx FPGA realm and it could be personalized and integrated in a new system image as well. The hardware which was used for the realization of the present project is the Ultra96v2 development board by AVNET 4. The only DPU version it supports is the one which is identified by the mnemonic DPUCZDX8G. Without delving into the details of naming conventions, this DPU instance is suitable for accelerating convolutional neural networks made up of a restricted but still very powerful subset of operators included in all the modern graph computation frameworks available for deep learning 5. Its single core nature, however, limits the capabilities of such piece of hardware when it comes to accelerating intrinsically parallelizable workloads such as batch computations over tensors.
The DPU has been included in a custom version of the popular embedded Linux distro Petalinux. This peculiar version is the PYNQ (DPU version) framework OS image, developed for enhancing productivity on FPGA exploiting the Python programming language. It eases the use of the programmable fabric providing an high level interface to the hardware, exposing, in particular, APIs for managing the **Overlays**. AN Overlay can be considered as a loadable module for managing and leveraging the programmable fabric which is easily usable thanks to a powerful ad hoc PYNQ module. Once the DPU overlay is in place, the IP is available for computations. At this point, it will run programs which are written according to DPU ISA. This is the point where **.xmodel** files come into play. They are binaries suitable for DPU execution. Such binaries are the result of a compilation phase carried out by the Vitis AI compiler, included in the Vitis AI toolchain.
Obtaining the .xmodel takes several steps. Depending on the preferred quantization strategy, eveything might start from a pretrained model (even though it would be extremely important to design the model thinking of the hardware) and then employing post training quantization or starting from a dataset and a model scheme and perform quantization aware training. The limitation of the second approach is that an highly personalizable computing platform is needed in order to carry out the highly specific training proce-

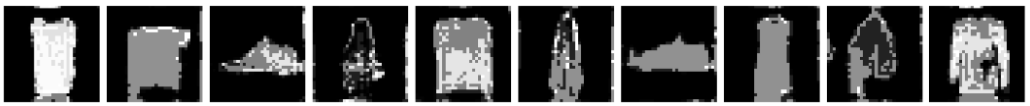


Figure 6: Images generated using the DPU.

dure. This was not the case during the development of this project, so post training quantization was used.
Before delving into the actual quantization, a word has to be spent on model design: the design space exploration made extensive use of a tool called **Model Inspector**. This instrument was used to infer the topology of the computational graph resulting from the quantization phase prior actually performing it. This latter process, in fact, partitions the overall graph in subgraphs which are suitable for DPU acceleration. The remaining subgraphs basically represent **islands of computation** which are to be ran over the CPU or possibly a GPU (if the embedded system at hand has one). Since no GPU is available on the Ultra96v2 board, CPU running is the only viable option for such subgraphs. This, however, lowers DPU utilization and can lead to performance drops, so the model hyperparameters were modified to fit the use case. This procedure was realized by means of trial and error but extra care had to be taken as modifying numerical hyperparameters of a neural network in arbitrary ways may lead to divergence of the training procedure or to a model whose quality is pretty poor. Convolutional networks, however, are quite simple, so accuracy was not hurt very much;moreover, the end result was perfectly suitable for DPU acceleration 9.
That being said, quantization has been realized using the Vitis AI model Optimizer. This tool executes two tasks: quantization and optimization (pruning). Unfortunately, the second one is not for free so it was not adopted in this project. As for the first, quantization involves calibration as well: this helps with the preservation of model accuracy.
Vitis quantizer produced the computational graph listed in 9. In the end, only the top of the network (tanh output) had to be run over CPU. One objective of these experiments was to obtain a model easily runnable over DPU and it was fulfilled: the .xmodel is not made up of fragmented subgraphs treatable as one unique computational graph with no need to manually schedule its subparts. Post quantization, the model was compiled to the .xmodel whose structure can be seen in the appendix 10.

Running the quantized network on DPU is two and a half times faster with respect to CPU.

```
Elapsed time for full image generation: 0:00:00.003560
```

Figure 7: DPU inference time.

```
[7]: from datetime import datetime
     i = tf.random.normal(shape=(1,128))
     start = datetime.now()
     out = np.tanh(quantized_model(i))
     print('Inference of quantized model (with the tanh output) on CPU takes ', datetime.now()-start)

     Inference of quantized model (with the tanh output) on CPU takes  0:00:00.075388
```

Figure 8: Test run of the neural network on CPU.

5 Conclusion: one last word

Adopting the development flow Vitis AI + PYNQ DPU has been extremely beneficial for optimizing the (fictitious) time-to-market of the solution. Going from the idea to the actual code has been an extremely fast process since high level tools where used to realize the imposed goal. This means that, even though the learning curve of FPGA is very steep, tools exist to ease the job for embedded deep learning engineers, who can, in turn, mainly focus on model development. However, the designer should always possess some basic knowledge of the embedded platform she is targeting. Some parts of the model development flow, such as the model design itself, have to incorporate prior information about the constraints that the chosen hardware platform imposes, otherwise **beautiful powerful models will end up being unusable** and this is the most frightening circumstance when dealing with hardware/software design. The same holds for highly efficient but not powerful enough models: if the model is incredibly fast and optimized but the accuracy and/or the quality of predictions is poor the result is not good. In a sense, the present work suffers from this latter problem: the model is pretty lightweight but the predictions quality of the quantized model is not very high. This is both due to the low power of the floating point starting model and to the adoption of post training quantization. This clearly dictates that model complexity could have been higher and this could have led to better results, provided that a smart pruning strategy had been used. However, two things have to be pointed out:

1. the design space exploration process is constrained by the capabilities of the DPU at hand (not every parameters combination works) and this complicates the network development and training phases

2. complex models require a powerful computing infrastructure to be properly trained, which was not available during the development of this project

That being said, some future work in the direction of experiments on Embedded Deep Learning with GANs might involve the study of pruning and quantization-aware training strategies for retaining as much prediction quality as possible. Moreover, it would be interesting to investigate power consumption and power savings (if any) when using FPGA for running generative networks with the above mentioned high level development flow.

6 Appendix

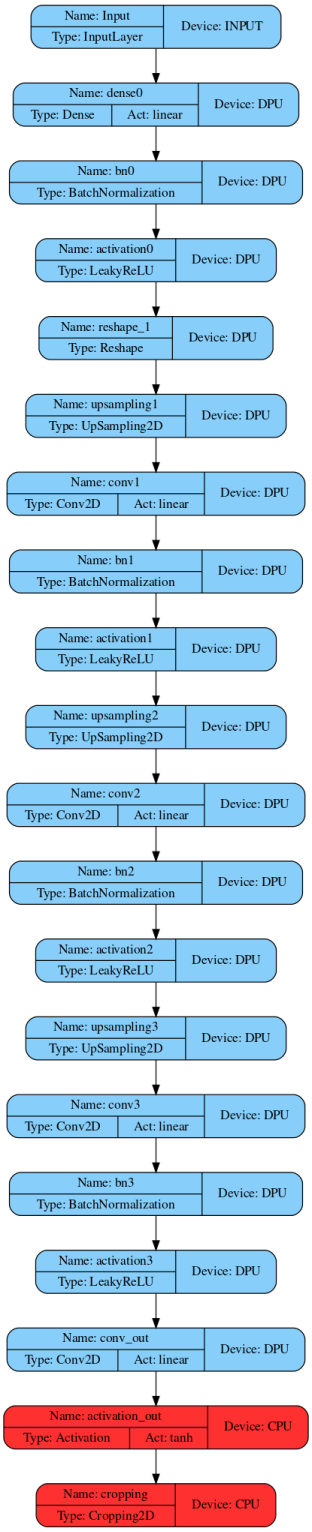


Figure 9: Result of the model inspection phase: the top of the network is not suitable for DPU acceleration.

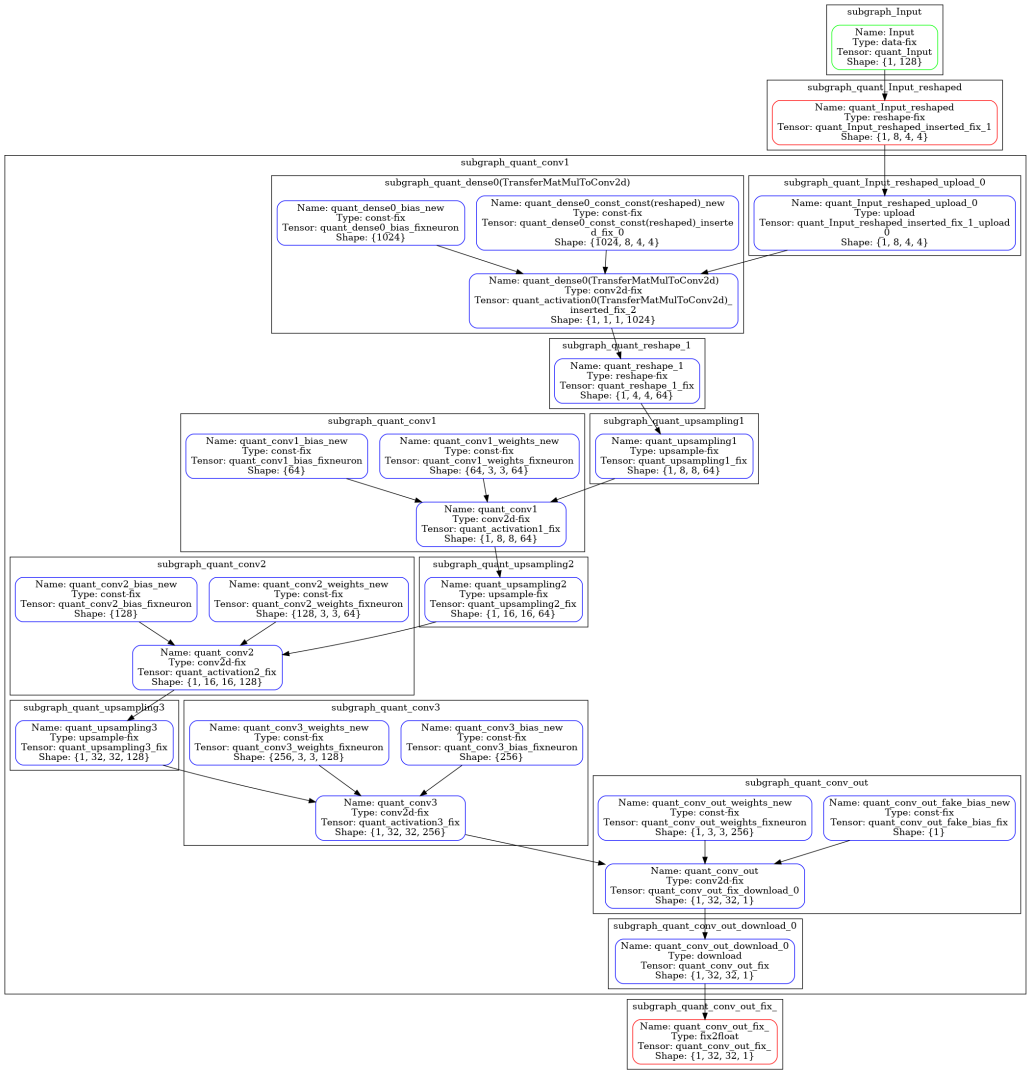


Figure 10: Result of the model compilation phase: full computational graph.