Space Tolerant CNN FPGA Deployment

Part 2: A Triple-Mode Redundant DPU

A. C. McCormick

Space Tolerant CNN FPGA Deployment, Part 2

This paper is the second part of a four part series of white papers providing an educational overview of the issues surrounding the deployment of Convolutional Neural Network solutions on FPGAs in a radiation susceptible environment. The first part documented a practical CNN processing core, which can be used to implement a wide range of CNN solutions. This second part, discusses the Space Hardening of this core, adding in Triple Mode Redundancy for radiation effect tolerance to control path circuitry. The third part will document the higher level control structures necessary to move data to and from the CNN core and dynamically reconfigure its operation to match the functional requirements of a part of a network. The fourth part of this series will document the deployment of this FPGA solution on the Alpha Data ADA-SDEV-KIT3 Space Development kit for the Xilinx XQRKU060 FPGA device.

Radiation Effects on FPGA Circuits

Radiation Effects on Integrated Circuits are widely understood and fall into a few key categories. Some result in failures that actually physically damage the device, others may require power cycling to resolve, while others can be mitigated dynamically. Total Ionising Dose is the level of radiation applied to the device over time that will cause it to fail through unrecoverable physical damage to the device. Single Event Latch-up is less severe, when a radiation event triggers a latchup of the circuit that typically requires the device (or part of) to be powered off to recover. Sensitivity to these effects is generally characterized for any Space Grade device, and often extra efforts within the design and manufacturing process are taken to improve the devices immunity to these effects.

With FPGAs however Single Event Upsets, where the logic level of a memory element in a register or SRAM memory location is flipped by the radiation energy, are a significant risk to functionality. In FPGAs this can have 2 significant effects. Firstly the SRAM on the device is used to hold the configuration of the FPGA circuit and any change to this will affect the circuit behaviour causing functional failure. Mitigating this requires the use of error checking and scrubbing solutions which continuously verify and correct the configuration SRAM. These can be most robustly implemented by an external device, but on-chip IP based solutions such as the Xilinx SEM IP might be a sufficient option is some lower radiation situations. The second effect is the generation of random logic errors generated within the FPGA logic circuit, when the SRAM used in the circuit has values flipped. Randomly flipping bits within the circuit will affect its functionality and behaviour, and it is mitigation against these affects that this paper will concentrate on.

Fault Tolerance of Artificial Neural Networks

Fault tolerance is a known property of Artificial Neural Networks such as CNNs. This is partly due to the inexact nature of the functions they learn to approximate, based on real life data, which can be noisy. There is also a degree of redundancy in most trained networks (sometimes optimized away using pruning). Therefore bit flips within the arithmetic of the CNN computation will have little or no observable effect on the classifcation or decision reached. However most CNN implementations are not direct models of artificial neurons, but are based around tensor processing cores, built out of multiply-accumulation units. Errors in moving the data around these circuits are likley to be far more significant as data might get dropped, or the whole computation might drop out of sequence, aligning the output decision with the data from the wrong input image. Therefore while there is some scope for relying on the inherent fault tolerance of the neural networks, effort to ensure the control plane runs error free is required.

Triple Mode Redundant Control Paths

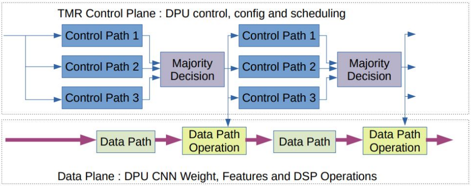

Figure 1 : Triple mode Redundant Control Path

One widely adopted approach to high reliability FPGA design in Space based electronics is triple mode redundancy. This technique effectively replicates the circuits in the design 3 times and then when a decision is required, the majority result at each significant stage is used. A single error on any path should be out-voted by the results from the 2 other paths. The disadvantage of this approach is that it requires 3 times the logic, computational resources and power of the basic design. However in this paper we are going to exploit the fault tolerance in the data path and only use the triple mode redundancy on the control paths in the design. Therefore the design will not require 3x the number of DSP48 multipliers or Block RAMs for weight storage, however it will require some additional logic for the data counters and other control and sequencing logic that tracks which product is being computed for each feature and weight in an image.

TMR Data Types and a Library of Functions

For a practical but easily readable and comparable implementation, the approach adopted to implementing TMR (triple mode redundancy) in this paper is to define new TMR data types in a VHDL package. These can be used in place of the standard VHDL types by simply changing the signal type, resulting in code that is very similar to the single mode source.

Within the example code, these new data types are captured in the package file *tmr_pgk.vhd*. A *tmr_logic* type is defined to replace *std_logic* with a 3 element array. The *tmr_logic_vector* type is defined to replace the *std_logic_vector* type widely used in the reference code. For numerical operations, the reference code used the *unsigned* type from the *ieee.numeric_std* library and therefore a TMR equivalent *tmr_unsigned* type is defined to replace these operations and signals.

The package defines a number of conversion functions between the TMR types and their basic single mode equivalent. Creator functions, such as *to_tmr_logic_vector* replicate the single mode type 3 times. Resolution functions (*tmr_resolve*) return the majority vote single mode type and other conversion functions such as *to_std_logic-vector* allow extraction of one of the 3 paths.

Operator overloading is heavily used for these types with standard operations such as addition, subtraction, multiplication and comparisons overloaded for the *tmr_unsigned* type to allow minimal changes from the reference code. Boolean logic operators *not, and, or* are overloaded for the *tmr_logic* type.

Higher level abstract data types can cause issues for synthesis tools such as Xilinx Vivado which does not seamlessly support 2-dimensional arrays such as those used for the *tmr_logic_vector* and therefore to allow synthesis and simulation of a synthesized netlist of the module, with TMR signals at the top level, additional flattening and unflattening functions are provided to convert each *tmr_logic_vector* to a *std_logic_vector* three times the size to give a top level with standard types to be used in the netlist.

With these functions in place converting the reference code to a TMR enabled version is relatively straight forward. In most cases the signal types are simply changed to the equivalent TMR type. At the top level configuration signals are fed in as TMR signals (flattened when necessary), as this will help preserve the 3 paths, and avoid any optimization that might occur. A few other changes are required to ensure that comparisons always compare signals of the same size, but in general the changes are not hugely significant and the TMR code resembles the original non-TMR code significantly. It is also very clear using this approach which signals are under TMR protection, in this case the control path signals, and which signals are not, in this case the data path for the weights and features of the network and the arithmetic operations and buffer storage used for the network.

This TMR package and all the modified VHDL modules can be accessed from the archive *onelayerdpu_tmr_v1_0_0.zip* and the code can be simulated and sytnthesized in a Vivado project that is built using the prj/mkxpr-1ldpu-cnn-tmr.tcl script: *vivado -source mkxpr-1ldpu-cnn-tmr.tcl*

Note that the large Caffe text definition file for the network *dk_tiny-yolov3_416_416_5.txt* is not duplicated in this .zip file and needs to be copied from the archive of source code for the first paper in the series: *onelayerdpu_v1_0_0.zip*

Simulating a Radiation Environment

Simulation of radiation effects on the FPGA can be achieved in a number of different ways. The approach used in this paper is to exploit the ability of the simulator to force signals within the design using the TCL command *add_force*. This command is specific to Xilinx Vivado simulation, however other simulators support similar commands. The TCL script *rad_sim.tcl* is provided to use this function to randomly generate bit flips on signals in the design. This is called by the *run_rad* function which has 2 parameters specifying the number of iterations and the running interval. This runs the simulation for the specified interval, and then calls the *gen_rad_event* function. This is repeated for the specified number of iterations. The *gen_rad_event* function also defined in the script reads the netlist of the simulated device under test, randomly selects a signal, with some exceptions, and then flips a bit in this signal for 15ns (i.e. more than 1 clock cycle).

The *run_rad* function can therefore be used in place of the TCL *run* command to advance the simulation while creating a number of random radiation like effects at the specified rate. For a quick demonstration of the issues the radiation might cause, a high number iterations and low interval should be specified. Using *run_rad* with the original reference design from part 1 of this paper series will quickly throw up errors in the processing, with incorrect amounts of data output the most likely error.

One limitation of this is that the *add_force* command does not support multi-dimensional arrays. Therefore using the behavioural simulation will actually give the *tmr_logic_vector* and *tmr_unsigned* types immunity from changes, which is unrealistic. Therefore to more accurately simulate the radiation effects on the triple mode redundant version of the design, the simulation should be performed as a post synthesis functional simulation.

Synthesis and Optimization Issues

When synthesizing TMR circuits one important consideration is the optimization the tools apply to reduce the logic used in a design. With a TMR design, there will be 3 identical copies of a circuit, and therefore it is very easy for an optimization algorithm to spot this redundancy and remove the extra logic, reducing the circuit back down to a single copy. The example code employs 2 strategies to avoid this. Firstly any input paramaters and signals at the top level that need replicated are actually fed in replicated, with the configuration values expected to be read in triplicate from external ECC DDR3 memory. This will avoid any possible redundancy optimization of these signals within the synthised module, as the tool cannot identify them as identical. More difficult to preserve are triple mode redundant registers defined deeper within the code however in these cases the replication is explicitly maintained using the *dont_touch* attribute in the VHDL source code which should disable the optimization for these paths.

Resource	Estimation	Available	Utilization%
LUT	4905	331680	1.47883
LUTRAM	83	146880	0.056508712
FF	13756	663360	2.0736854
BRAM	86	1080	7.962963
DSP	130	2760	4.710145
IO	125	624	20.032051
BUFG	1	624	0.16025642

Table 1 : Reference DPU Resource Utiliziation

Resource	Estimation	Available	Utilization%
LUT	17634	331680	5.31657
LUTRAM	83	146880	0.056508712
FF	24139	663360	3.6388988
BRAM	86	1080	7.962963
DSP	134	2760	4.8550725
IO	307	624	49.198715
BUFG	1	624	0.16025642

Table 2 : TMR Design DPU Resource Utiliziation

Tables 1 and 2 show the resource utilization for the original reference DPU core and the TMR design. This shows that the TMR does increase the size of the core, in terms of LUTs by a factor of 3 and FFs by a factor of 2, however it does not result in a significant increase in the more scarce BRAM and DSP usage, and therefore by only applying TMR to the control path and not the data path implementing the fault tolerant neural network computation, the requirement for resources does not increase across the board by a factor of 3.

Conclusions and Next Steps

This part of the paper has covered the subject of radiation effects on a Space deployable CNN implementation. Common radiation effects on FPGA circuits have been discussed. The Triple Mode Redundancy mitigation technique for mitigating soft single event upsets in the processing has been explored. The fault tolerance and redundancy in the CNN weights and calculations has been exploited, to provide a circuit where only the control plane signals get protected by TMR as bit errors in the CNN arithmetic operation are deemed as acceptable, and unlikely to significantly affect operation. By using this approach, the resulting rad-tolerant circuit provides reliable operation but without the cost of tripling the use of every resource on the FPGA deployed. This approach has been tested in simulation, exploiting simulator level scripts to force single errors on signals within the circuit.

The first 2 papers in this series have concentrated on implementing a core IP block for CNN operations. The next paper will concentrate on how to connect up this IP core to external resources such as DDR memory used to store weights, features and intermediate results. The next paper will also cover the scheduling and control of the DPU, using a state machine, that can read a sequence of descriptions from memory, which describe the DPU configuration, the location of weights and data in memory and use these to control the DPU and transfer the required data.

Revision History

Date	Revision	Nature of Change
31/05/21	1.0	First draft