POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Analysis of faults and errors moderation for deep learning operators executed on GPUs

Computer Science Engineering Project
Computer Science Engineering - Ingegneria Informatica

Filippo Balzarini, Matteo Bettiati
10719101, 10717078

Advisor:
Prof. Antonio Miele

Co-advisors:
Prof. Luca Cassano
Dott. Alessandro Nazzarri

Anno accademico:
2022-2023

Abstract: Analysis of faults and errors moderations for deep learning's operator executed on GPUs is a project that consists of some specific experiments on GPUs to analyze how hardware errors could affect software operations. It's part of CLASSES[1] (Cross-Layer AnalysiS framework for Soft-Errors effectS in CNNs) which is our main landmark for all the choices we made.

Contents

1. Introduction

In those years we are facing a growing interest in employing convolutional Neural Networks (CNNs) for perception functionalities in a wide range of application domains, including safety- and mission-critical ones. In this context, the CNNs are generally executed on Graphic Processing Units (GPUs), because of its Single Instruction Multiple Data (SIMD) architecture, capable of speeding up the highly data-parallel elaborations that characterize these applications.

Many applications of the CNNs require very high reliability since it has been demonstrated that soft errors, such as Single Event Upsets(SEUs) may interfere with the functionality of electronic systems.

CNNs applications have been reported an intrinsic degree of fault resilience due to several reasons: i) dealing with noisy inputs or data quantization, ii) their output may be probabilistic estimates, or iii) produced data may be used by humans, whose perceptual limitations provide resiliency to a certain level of inexactness.

The core of the project is to analyze how the CNNs react to faults injections to answer the question "Is the downstream system able to correctly carry out its task with the produced, possibly corrupted, output?".

The injection analyzed in this project has been generated at the architecture level. Then, the corrupted tensors generated by the CNN have been analyzed and classified, in order to predict the result of a soft error, using Classes[1] it can also be analyzed how the CNN result is influenced by the corrupted tensor. Furthermore, the data are organized in three different files so that the statistics can be studied from different points of view. The first one is a file that contains all the type of errors and their probability of occurrence. The second one is a spatial model, the errors are grouped by the times they have appeared in the corrupted tensors and it specified the type of error meanwhile the third one, the count model, is grouped in the same way but it specified their occurrence in the corrupted tensors. Finally, we provided a tool useful to merge all the results of the various experiment to have an overall view of the distribution of errors to be aware of the effects the errors are going to induce.

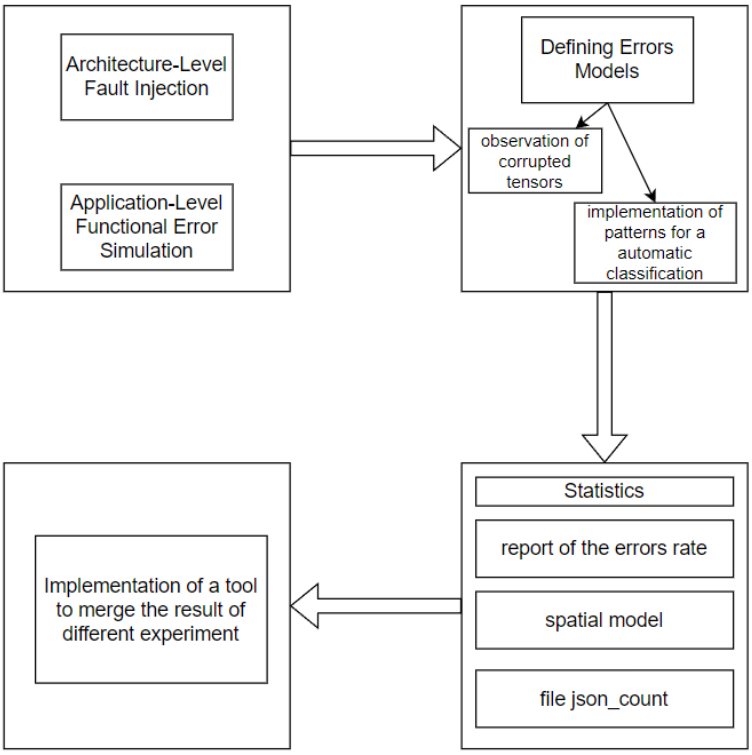


Figure 1: The process is divided into four stages: Fault Injection, Error Modeling, Collection of statistics, and the merge of the experiments' statistics

2. Background

This section provides a brief background introduction of CNNs, GPUs, and the fault model, which are the key elements of this context.

2.1. CNN

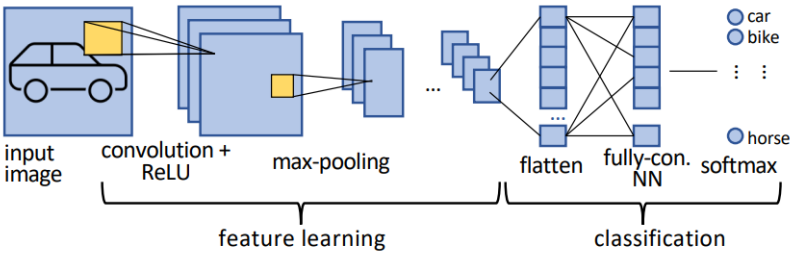


Figure 2: The typical topology of a CNN is described in terms of a sequence of layers for features learning and a final classification layer, integrating a fully-connected Neural Network.

A Convolutional Neural Network is a Deep Learning model generally employed in image processing and computer vision to derive a semantic representation from the input images to accomplish a high-end task, such as item classification, object detection and image segmentation.
A CNN is internally organized in a sequence of layers, each one processing multidimensional data, known as *tensors*, by means of *operators*. A tensor consists of a multi-dimensional stack of bi-dimensional matrices of values, called feature maps, generating a multidimensional grid. Taking for example an image, it can be seen as three stacked feature maps, each one for a color channel, thus producing 3D tensors. There are several operators, that can be grouped into the following classes:

- *Convolution*, used to "learn" and extract features from the input by inferring the appropriate weights;
- *Batch normalization*, used to fix the data distribution, speeding up the learning process
- *Activation function*, a mathematical function applied element-wise to mimic the biological activation of a neuron.
- *Max-pooling*, used to reduce the size of the tensor for increasing the degree of generalization;
- *Element-wise operators* for classical math operations or single-element manipulation, such as addition, multiplication and so on.

Since operators are general in the size of the input/output, they are characterized by a set of hyper-parameters, specifying the actual size of the processed tensors. Each operator, organized in a sequence of layers devoted to feature learning, takes in input and produces output tensors of a specific size, based on the structure of the CNN. The final result produced by the feature learning is a tensor as well. When the application goal of the CNN is the *classification*, the CNN contains a second part, where the final tensor is flattened and fed into a fully-connected Neural Network and a softmax function, in order to produce the set of probability, describing the likelihood of the identified object to be classified according to the set of classes.

2.2. GPU

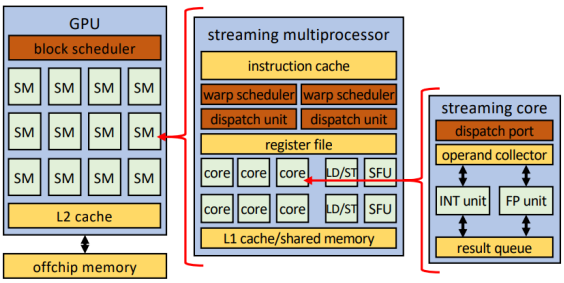


Figure 3: NVIDIA GPU hierarchical architecture

A GPU is a many-core organized in an array of Streaming Multiprocessors (SMs). The SM is in turn structured following the SIMD paradigm, thus with a single control unit scheduling and dispatching instructions, and a grid of simple arithmetic cores, memory load/store units and other special functional units for math operations, nearly streaming cores. Then, each single streaming core is internally implemented to read data from the register file, execute either integer or floating point operations and store results in the register file.

A *Kernel* is a software function accelerated on the GPU by means of a large grid of threads grouped in a number of blocks. Threads compute on separate portions of data and each threads' block is dispatched by the block scheduler to a single SM for its execution; the group is then subdivided into *warps*, executed with the SIMD paradigm; indeed the SM contains multiple warp schedulers and dispatch units to execute concurrently several warps. Branch instructions where divergences may occur are executed by means of predicated approach; all alternative branches are executed in sequence for the entire warp, and every single thread is active only in the selected branch and disabled in the others. Finally, the dispatching unit schedules several interleaving warps to maximize the SM throughput.
The GPU has a memory hierarchy organized in an off-chip global memory and two levels of caches, a unified L2 and a per-SM L1, in addition, it contains a shared memory, used internally by the threads of a single block in order to cooperate and synchronize without delays.

2.3. The fault model

We consider the Single Event Upset (SEU) fault model, whose effect is a bit-flip in a value stored in a register of the computing platform. Because the faults are rare events, the assumption of single faults holds very well. Faults may cause the application:

- to crash or raise exception from the operating system that blocks the execution
- to hang leading to a non-termination
- to terminate by product an erroneous result, without any alert, this is called *Silent Data Corruption (SDC)*

The SDC, different from the first two types, can not managed at operating system level, so it represent the most critical situation, that needs to be analyzed. In a GPU the SDC may corrupt the execution of a single thread within the same warp belonging, to a single kernel. In case the shared memory is used, erroneous data produced by a thread can be propagate among several threads of the same block. However, memories and cache are hardened by means of Error Correction Code (ECC); therefore, no fault in those locations will be here considered.

For fault injection we used nvbitfi[2]. NVBitFI provides an automated framework to perform error injection campaigns for GPU application resilience evaluation. NVBitFI builds on top of NVIDIA Binary Instrumentation Tool (NVBit), which is a research prototype of a dynamic binary instrumentation library for NVIDIA GPUs. NVBitFI offers functionality that is similar to a prior tool called SASSIFI.
The fault injections have been made in several ways, both for instruction injection or simpler value injections, such as:

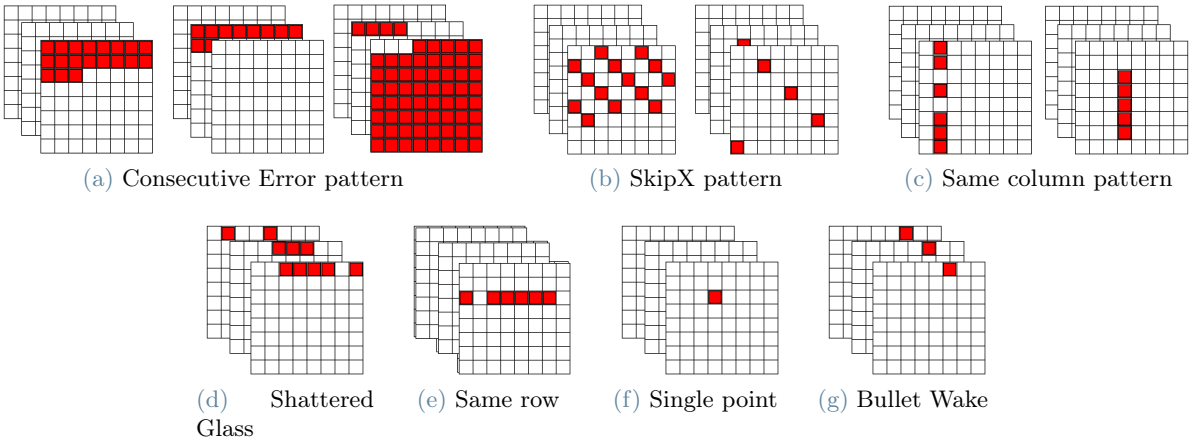
- fp: floating point instruction
- gp: general purpose extraction
- ld: load instruction
- rv: injection random value
- fsb: injection flip single bit

3. Define Errors Models

This section presents the result of the application of the methodological framework in the definition of the error models. We analyzed thousands corrupted tensors to identify the recurrent corruption patterns. In the following we report the results of such analysis based on the previously discussed aspects of interest (spatial distribution), followed by the definition of the error models.

3.1. Spatial Distribution

We analyzed the spatial distribution of the erroneous values within the corrupted output tensor. A first classification is between faults that produce one or more errors in a unique feature map and those whose effect spreads over multiple feature maps.



(a) Consecutive Error pattern (b) SkipX pattern (c) Same column pattern

(d) Shattered Glass (e) Same row (f) Single point (g) Bullet Wake

Figure 4: Examples of the found error pattern.

Single Feature Map Most of the faults that cause less than 16 erroneous values in the output tensor affect only a single feature map. This is due to the fact that the GPU kernel is implemented to group threads working on the same feature map in a single block; thus such multiple errors may be due to the corruption of predicated instructions. In this subfamily of spatial distributions we identify:

- *Single point* [4f]: a single value is corrupted.
- *Same row* [4e]: multiple corrupted values lie in the same row.
- *Same column* [4c]: multiple corrupted values in the same column.
- *Random*: no regular pattern.

Multiple Feature Map Most of the faults that cause more than 16 erroneous values in the output tensor spread the corrupted values among several feature maps; this is due to the fact that, as mentioned, such operators heavily exploit shared memories. In this subfamily of spatial distributions we identify:

- *Bullet Wake* [4g]: the same location is corrupted in all (or in multiple) feature maps.
- *Shattered glass* [4d]: like one or more Bullet wake errors, but in one or multiple feature maps the corruption spreads over a row (or part of the row).
- *Consecutive* [4a]: serial values corrupted, without correct values in between. Corrupted values can occupy more than a rows.
- *SkipX* [4b]: the distance between corrupted values is always composed by X correct values. The X value it's the number of correct tensors between the first and the second corrupted values. The distance between the first corrected values and the first corrupted it's negligible.
- Random: no regular pattern.

Those error classes have been individuated due several injections and analysis of the results, the corrupted resulted tensors has been compared with the correct result, obtained without the fault injection. We considered that a value in the tensor has been corrupted only if

$$\Delta E = |V_{ct} - V_{gt}|, \tag{1}$$
$$\Delta E > 0.001 \tag{2}$$

given V_{ct} as the value of the corrupted tensors and V_{gt} as the value of the golden tensor (the one that is not been corrupted).
Once the injections have been made and the error classes has been individuated, we created a script[3] able to classify the corrupted tensors in the right class written in python.

3.2. Automatic classifier

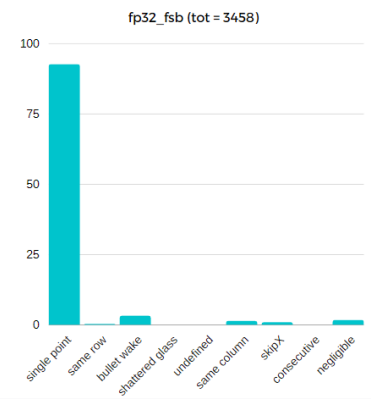
In this section we are going to describe, at high level of abstraction, the core idea of the python script able to classify the tensors from an experiment. The script[3] has three core functionalities:

- automatize the classification of the pattern
- provide a spatial and a count statistics model

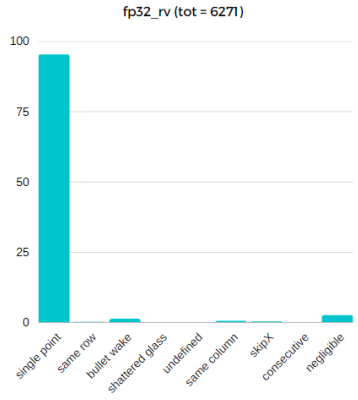
The input of the script is a folder containing all the corrupted tensors of the injection and the correct output tensor, the output consists in a spatial and count model in json format and a directory, containing text files, which the name correspond to an error pattern and contains all the name of the corrupted tensor of that experiment.
The script starts from the assumption that the analyzed tensor is simultaneously part of all the patterns. During the analysis, if an error occurs in a coordinate or in a channel that cannot be part of a specific pattern, that pattern is excluded from the possible ones. Once the iteration is completed, the tensor will be assigned to only one pattern, it could happen that one tensor can be associated to two pattern, but a simple hierarchy will be applied, i.e. a *consecutive error* could be recognize as also a *same row*, but, because the *same row* class is more restrictive, the tensor will be assigned to the *same row* class.
Once the tensor is classified, we proceed to put the file that contains it in a proper directory with all the others that has the same pattern of error.
Finally the scripts generates some files to collect statistics from different point of view.

4. Statistics

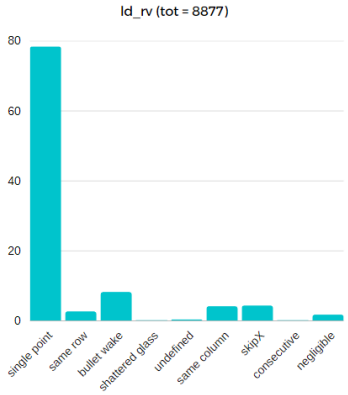
The script also collect the load of data in order to create some statistics file and report the result of the injection on the corrupted tensors.
Precisely it generates three different type of files.
The first one it's a csv file and it's generated by a small script we made called *writeData* which can be found in the CNN Error classificator repository[3]. The file created contains the result of an entire experiment. All the data are separated depending on the injection.
Down here you can find all the results of the experiment sorted by type of injection.



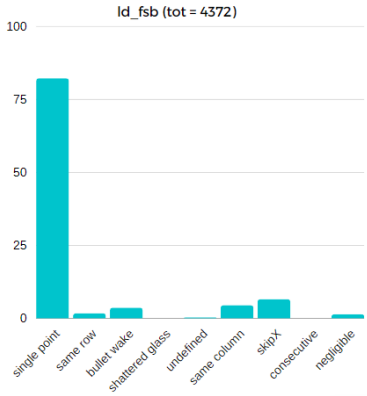
(a) fp32_fsb



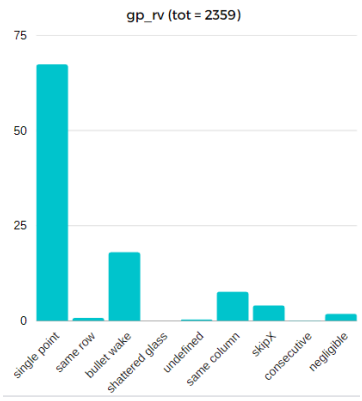
(b) fp32_rv



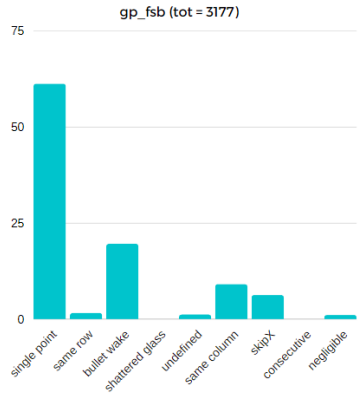
(c) ld_rv



(d) ld_fsb



(e) gp_rv



(f) gp_fsb

Figure 5: This statistics report the type of errors of all the experiment we have done merged together.

4.1. Spatial Model

File {operator}_spatial_model.json contains information about the spatial patterns. The keys of the json represent the cardinality for which we are describing the pattern, and the value connected to each key is another dictionary. If the key is "1" then this dictionary contains only the following data "RANDOM": 1 since with only one corrupted parameter the concept of spatial pattern has no particular meaning.
Instead if the key is greater than one the dictionary will have two keys: FF and PF. Associated to the key **FF** we have a dictionary where each key has a possible value between 1 and 8 that represents the eight possible patterns that we observed. The values here are the probabilities of occurence for each pattern. Associated to the key **PF** instead we have the description, for each pattern, of the most common dispositions. For a better understanding of this concept let's observe an example.
Let's suppose that we have 5 corrupted values, the relevant information is the following

Algorithm 1 Example of a spatial model

```
"5": {
    "FF": {
        "0": 0.5578512396694215,
        "4": 0.44008264462809915,
        "6": 0.002066115702479339
    },
    "PF": {
        "0": {
          "(0, 1, 2, 3, 4)": 0.13333333333333333,
          "RANDOM": 0.8666666666666667,
          "MAX": "15"
        },
        "4": {
          "RANDOM": 1,
          "MAX": "15"
          },
        "6": {
          "((0, (-8, 0)), (2, (0,)), (4, (-8, 0)))": 1,
          "RANDOM": 0,
          "MAX": [0, 0, 0, 0]
        }
    }
}
```

Here we can see that the patterns we observed for this cardinality are

- 0, or *Same Row* [4e]
- 4, or *Bullet Wake* [4g]
- 6, or *Shattered Glass* [4d]

If we select the pattern *Same Row* we see two possibilities under PF. The first one, with a probability of 13%, is associated with the stride vector (0, 1, 2, 3, 4). This means that we will randomly select the position of the first corrupted value and then the other four values will be at a distance of 1, 2, 3, and 4 places respectively. The second possibility is to randomly select the disposition of the five points. In this case we must keep track of the MAX value that represents the maximum distance between two consecutive corrupted values.

4.2. Count Model

File {operator}_anomalies_count.json contains information about the cardinalities observed. The keys of the JSON are the cardinalities, and the associated values are lists of two parameters. The first is the number of times we found that specific cardinality and the second is its probability of occurrence.

5. Merge Statistics

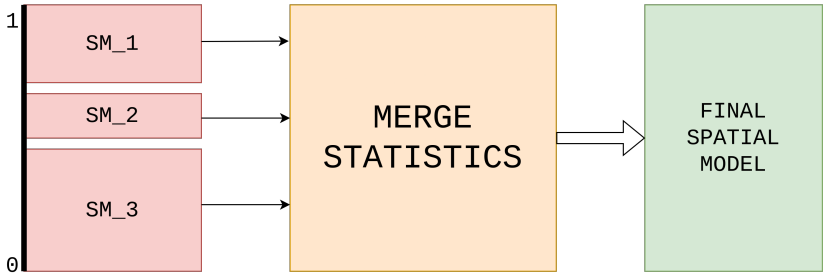


Figure 6: block diagram that describes the core idea of the merge statistics script

As already mentioned, the automatic classification of an experiment creates a spatial and count statistics model. Merge Statistics6 is a script able to merge the results of spatial error models given a set of spatial models and their weights. Using merge statistics, different results can be melded together in order to obtain a single spatial model that describes the correct distribution of the error classes in the experiments. The script is useful for sticking together different experiment results without losing information.
Weights should be chosen, for example, based on the dimension of the experiments, given two experiment results, if the first experiment collected data from 200 corrupted tensors, and the second experiment, collected only 100 corrupted tensors, it would be useful to assign 0,67 (200/300) to the first experiment and 0.33 to the second.

6. Conclusion

The project's greatest strength is that thanks to the analysis made up here, we can predict with a close approximation which damage can a hardware error inflict on the software.
The idea behind the project is to analyze a large amount of data in order to be capable to prevent soft errors or at least know the damage they could create.
The script also confirmed all data provided in the first version of CLASSES and thanks to the Merge Statistics tool, the use of CLASSES is now more fluid. Future work will bring further optimization to the script that classified errors, bringing a completely autonomous algorithm for the classification of errors that doesn't just classify errors with known patterns, but will be also able to discover new patterns itself.

References

[1] D4De Design 4 Dependandability. Classes https://github.com/d4de/classes, 2020.

[2] NVlabs. NVBitFI: An architecture-level fault injection tool for gpu application resilience evaluations https://github.com/nvlabs/nvbitfi, 2022.

[3] Filippo Balzarini and Matteo Bettiati. CNN error Classificator https://github.com/filomba01/cnn_errorclassification, 2023.