README

We provide this README file to illustrate in detail on the implementation process of GraphFI.

**Setup:**

We deploy GraphFI on a host Intel i7-7700 CPU (2.60 GHz) with 16 GB memory and an NVIDIA GeForce RTX 3090 GPU. We use a cycle-accurate, open-source simulator GPGPU-sim (v4.2.9)[1] to obtain the dynamic instruction information.

**Usage：**

Below, we provide a detailed description of the implementation of GraphFI, including (1) the investigated graph applications and dataset, (2) Graph application and data topology profiling module, and (3) Fault injection Simulation module.

1. ***Graph applications, datasets, and output quality metrics***

To evaluate the effectiveness of GraphFI, we investigate four graph applications which are widely used in literature. We perform evaluations on four real-world graphs (i.e., FD[2], GP[3], FBP[3], OF[3]).

* SSSP: Single Source Shortest Path
* SCC: Strongly Connected Component
* PR: Pagerank
* HITS: Hypertext Induced Topic Search

Incorporating the domain-specific knowledge and the reference to previous error resilience studies, we adopt Top-k ranking for PR and HITS as users typically only care about the correctness of the top-ranked items. For SSSP and SCC, we use Relative L2-norm to estimate the deviation of the overall graph’s paths/connectivity from the ground truth. Based on the quality metrics, we divide the fault injection results into three categories:

* Benign Corruption (BC): The graph application output exactly matches the fault-free output or only has negligible output quality degradation that is acceptable to the user/system.
* Severe Corruption (SC): This indicates the program executes without noticeable symptoms while there exists unacceptable mismatches between the program output and the fault-free output.
* Detected: This indicates the error-incurred symptoms such as hardware traps or crashes.

1. ***Graph application and data topology profiling module***
2. ID-GraphFI:

We first modified the output options of **GPGPU-sim** to obtain the **dynamic instruction set** along with the **corresponding registers**. Then, we filtered the address calculation instructions from the dynamic instruction set (whose registers are “RD” type) and compute the address calculating instruction ratio of each iteration. This is the basis for clustering Detected-case similarity iteration groups. As this ratio typically remains stable, we randomly select individual iteration to represent the overall Detected rate of the program.

Then we run **"kernel.cuh"** to obtain the active vertex set of each iteration. We divide consecutive iterations with similar active vertex sets into an SC-case similarity iteration group, and select a representative iteration from each group as FI candidate to efficiently evaluate the program SC rate.

1. TD-GraphFI:

After selecting representative iterations, we then identify resilience-similarity vertex groups in each selected iteration. We utilize the correlation between vertices' error resilience and their neighborhood similarity/clustering properties to identify vertex communities in which vertices share similar neighbors (N-Communities) or (b) in highly clustered subgraphs (C-Communities). From each set, we select one representative vertex for fault injection.

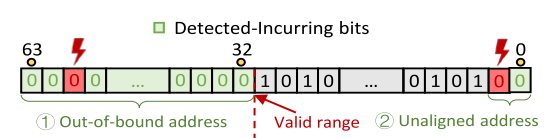
Running the **"similar.py"** program generates the neighbor similarity set (N-Communities).

Running the **"cluster.py"** program generates the high-clustered set (C-Communities).

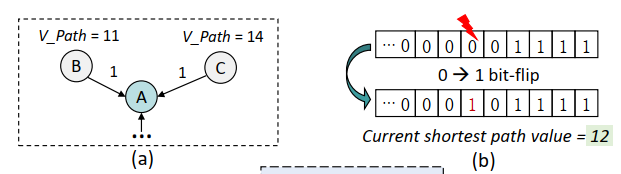
1. MD-GraphFI:

Each selected vertex (i.e., thread) execute several instructions. We divide them into Detected-related instructions (i.e., address calculation instructions) and SC-related instructions (i.e., numerical calculation instructions).

For Detected-related instructions, we obtain the base addresses of involved arrays (such as through a pointer variable in CUDA). Then, the valid address range can be estimated based on the size of the array elements and the thread's index offset (calculated by thread ID and block ID), which allows us to identify the higher bits that may cause out-of-bound memories (i.e., Detected cases). We obtain the address register value through GPGPU-sim, and based on this valid address, we can estimate which bits in the address register will lead to detected errors, as shown in the figure below.



Due to certain graph update operations have fault masking properties, as shown below, such as in SSSP, where the min() function selects the shortest path from all in-edge neighbors and updates the target vertex path accordingly. Faults in the higher bits of non-shortest path operands may be masked by the min() function.



For SC-related instructions, we first exclude the “abnormal” register bits whose error-incurred outcome is surely BC according to the fault masking property of the graph updating operation (if there is). The remaining register bits can be handled by binary search fault injection based on monotonicity.

1. **Fault injection simulation module**

We can obtain the representative iteration, vertex (i.e., thread), and register bits from graph application and data topology profiling module. Based on this, we perform fault injection experiments. Running the script **"inject\_one.sh"** allows for a specified number of fault injections. The main file for fault injection is **"fault-inject.py"**, where faults can be injected into **specified iterations, threads, and instruction register bits** by tunning the corresponding configuration in **"fault-inject.py"**. The outputs of the fault injection are **"outcome.txt"** and **"basic.txt"**, which record the results of fault injection (i.e., BC, SC, and Detected) as well as detailed information for each fault injection experiment.

Citation:

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda

workloads using a detailed gpu simulator. In 2009 IEEE International Symposium on Performance

Analysis of Systems and Software, pages 163-174, 2009.

[2] “Foldoc,” http://vlado.fmf.uni-lj.si/pub/networks/data/dic/foldoc, 2024.

[3] “Grid plant, fb pages, and open flights,” https://networkrepository.com/, 2024.