Abstract   
General-purpose GPU computing has become a major high performance parallel computing platform due to the many-core environment that facilitates an extraordinary computational power. However, GPU programming bug patterns have not yet been well explored.

In this paper, we perform a study on GPU program bugs. We analyze 241 GPU program bug samples found in GitHub archive. We identify the common underlying root cause of the bug in the GPU program, the strategies used to manifest the buggy behavior, and the fixing strategies used to remedy the bugs in the GPU programs. The finding made in this work can provide a foundation for the development of detection and prevention techniques for bugs arising in GPU programs.

1. INTRODUCTION

The wide-spread of parallel computing architectures like many-core and multi-core systems that increase hardware performance and improvements to the programmability, parallel programming is used in many applications, especially for scientific computing. The general-purpose GPU programming facilitates extraordinary computing power because of the amount of parallelism available in GPUs to execute applications quickly.

However, the testing the correctness of GPU programs is a challenge due to its undefined behaviors in platforms like CUDA C which inherits problems in C that bring new challenges to GPU programming [R6]. Programming bugs refer to errors that lead programs to misbehave. In GPU programming, users have experienced such buggy nature while executing the GPU programs. In the past few years, researchers have increased efforts to address problems related GPU programming [R10, R8, R6]. However, existing research on GPU programming bugs mostly focus on a specific area such as memory bugs, synchronization bugs, compiler bugs and many more. Therefore, our approach is to investigate the linkage between the root causes, manifestation strategies, and fixing strategies that can further understanding further understanding of GPU bugs and how to fix them in a systematic way.

Investigating the types of GPU programming bugs can help he developers to find solutions and fix the issues in their applications. This can also help the community to explore research in finding automated bug detecting strategies for GPU programs. Therefore, researching GPU program bugs can help developers by providing insights on effective detection and prevention methods.

To further investigate GPU program bugs, we collect and analyze 241 real-world GPU program bug test examples found in projects using GitHub archive. For each bug, we inspect commit descriptions, issues reports, reported causes, and changed code. We focus on the following questions and summarize our findings and implications in Table.

RQ1: What are the typical root causes behind bugs in GPU programs?  
We examine the collected GPU program test samples to determine the underlying causes of buggy behavior. We group similar root causes together into 8 main categories: *Memory Issues, Environment Issues, Synchronization Issues, API function flagging Issues, Floating Point Issues, Compiler Issues, Library Issues,* and *Visual or Graphic Issues.*We categorize memory related issues in GPUs and GPU programming under the *Memory Issues* category that causes the programs to fail. Some GPU programs fail due to different versions and platforms that is being used to execute these programs. Therefore, these issues are categorized into *Environment issues*. GPU programming helps synchronization between memory copies, threads, and streams. The issues related to this category are separated into *Synchronization issues*. Some repositories show errors in API functions because of the incorrect use of them. Therefore, we separate these issues under the category of *API Function Flagging Issues*. Some programs fail during the compilation because of the issues in the makefiles, NVCC compiler, and compilation errors. Therefore, we categorize these issues into *Compiler Issues*. GPU programming supports different libraries. We found that some programs fail when using these libraries. We separate them into *Library Issues* because it is easy to find the library and the related issue if any developer comes across with it. GPU programming supports single and double precision floating point calculations. The programs that fail when working with floating points are categorized into *Floating Point Issues. Visual and Graphic Issues* are related to failures in scaling when implementing graphics using GPUs.

RQ2: What conditions do bugs in GPU programs manifest and how they are reproduced? In order to understand how users, report intermittent behaviors, we investigate the common strategies used to manifest the GPU program bugs in the test samples. The data reveals 5 strategies used to reproduce and report bugs in GPU programs: *Specify memory limitations, Force Environmental Conditions and Configurations, Provide Code Snippets, Check for Concurrency,* and *Specify Problematic Platform and Versions.*

RQ3: How are these GPU programming bugs typically fixed?   
We identify the bug fix applied to each collected GPU program test sample and group similar ones together. We find 5 main categories for bug fixing strategies: *Fix Memory, Dependency, Alter Programs, Add GPU Support and Configurations,* and *Fix MakeFiles.*

We investigate the impacts that these GPU programs have on the bugs, and we find several distinctions. Based on the investigation of above research questions, our main contributions of this study are:

1. Our study provides guidance for developers to create reliable and stable GPU program test suites, which can reduce the occurrence of bugs in GPU programs.
2. Our study summarizes the commonly-used manifestation and fix strategies of GPU program bugs to help developers easily reproduce and fix the bugs, allowing them to avoid wasted development resources.
3. Our study motivates future work for automated detection and fixing strategies of GPU program bugs.
4. BACKGROUND
5. Impacts on GPU program bugs
6. Individual Test Failures: The simplest impact on a bug can have on a GPU program is that the individual test run will fail. This buggy behavior leads to minimal amount of time and resources wasted by attempting to retry the single test.
7. Build Failures: The GPU program bugs can lead to build failures that cannot perform the program’s intended job. The bugs in this stage lead to wasted time trying to identify the underlying cause of build failure to find out that the failure was not caused by the code.
8. Timeout failures: Some GPU program bugs does not cause tests to fail outright. Instead, they cause hangups that lead to timeouts. These hangups waste the time as the system waits without finishing until a certain specified timeout is met.
9. METHODOLOGY
10. Sample Collection

In order to collect samples of GPU programming related issues, we retrieve commit samples from GitHub repositories. First, we obtain a list of GitHub issues mentioning keywords ‘GPU’ and ‘GPU programming’ using the GitHub search API dated from January 2020 to December 2020 to identify 118,438 issues posted. From this set of issues, we download all of them for further evaluation. Next, we search the issue types for patterns related to ‘GPU\*’, ‘memory’, ‘threads’, ‘streams’, ‘environment’, ‘synchronization’, ‘concurrent’, ‘multi-GPU’ and ‘compile\*’ to find issues related to GPU programming. This step reduces the number of commits to 1,581. In order to confirm which of these issue links were related to GPU programming related, manual inspection was performed on the issues in the list. After manual inspection, and removing duplicate issues, the number of verified GPU programming bugs is 241.

1. Portion of GPU Programming bugs to other bugs

We find that GPU programming tests collected in our methodology make up a small portion of all tests available in these repositories. Based on GitHub archive, the number of issues that captures the keyword ‘GPU’ consist of issues that is not relevant to GPU programming. Also, we find that some of the open issues that are solved still exists as open issues. An example is seen in the [Open1] where is addresses an issue related to illegal memory where the issue provides a possible manifestation and a fixation. Therefore, we include such open issues in our study. Also, there are some issues that merge another GitHub issue as a solution for the bug.

1. Sample Inspection

After collecting these issues of GPU Programming bugs from GitHub, we manually inspect the collected samples to identify information relevant to our research questions. In particular, we analyze the collected issue reports by inspecting the issues for following traits: the root cause of the bug, how the bug is manifested, and how the bug was fixed. For the issue reports, we inspect the developer comments and the linked commits. Through the inspection we obtained the sample set of 241 GPU programming bug issues.

1. Dataset Composition

Our dataset consists of a diverse of GPU programming bugs. The languages of the GPU programming bugs analyzed are CUDA, OpenCL, and GPU-accelerated computing with Julia, python.

IV. CAUSE OF BUGS

We investigate the collected GPU Programming issues to determine the root cause of the bugs. We manually inspect the related issues of the test in order to locate the code or condition that caused the bug. We base our root cause categories as Memory, Environment, Synchronization, Compiler, Math Errors, API functions or flagging issues, Library, and visual or graphics issues. The categorization results are summarized in Table.

TABLE: Summary of Root Cause Categories Found.

|  |  |  |
| --- | --- | --- |
| Root Cause  Categories | Root Cause  Subcategories | Total |
| Memory | Out of memory  Illegal Memory Access/ allocation  Memory Copies Memory Usage | 35 23 10 10 |
| Environment | Driver issues  Version Issues  Devices Issue/ Environmental Variable | 8  27 17 |
| Synchronization | - | 37 |
| Compiler | Make Files  NVCC compiler  Compile | 5 11 7 |
| Floating point issues | Floating point | 5 |
| API function/ Flagging Error |  | 12 |
| Library | Math library cuRand NCCL Thrust | 9 3 11 4 |
| Visual/ Graphics issues | - | 6 |
| Other | - | 1 |

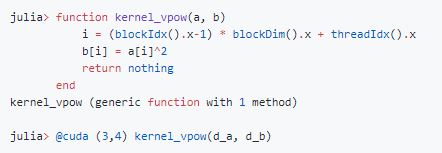
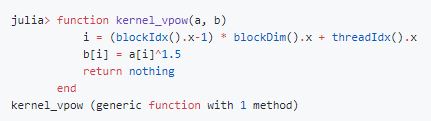
1. Categorization

After manual inspection of the bug issues, we identify eight categories that root causes of bugs in these tests can fall under: (1) Memory issues, (2) Environment Issues, (3) Synchronization issues, (4) Compiler issues, (5) Math Error issues, (6) API function or Flagging issues, (7) Library issues, and (8) Visual or Graphics issues.

1. *Memory Issues:* We have found the root cause for a significant portion (33%) of GPU Programming bugs analyzed arise from issues related to the memory. The common cause for this issue is allocating memory that is larger than the available memory in the kernel. Among these memory issues we, we identified four subcategories that group similar root causes together.
2. *Out of Memory issue:* GPU programming issues related to this category attempt to allocate the device memory that exceeds the available memory. The access to larger block sizes can lead this problem. An example is seen in the [2] when computing a Jacobian matrix leads to an out of memory issue because of the large block sizes. Another example is seen in [1] when a deep learning model is trained in the GPU, the CuPy memory allocation in the CUDA is not sufficient to train the model and leads to an out of memory issue.
3. *Illegal Memory Allocation/Address issue:* GPU programming issues related to this category mostly occur during the runtime when the program access illegal memory.   
   Most common mistake is wrong index computation for arrays or wrong array offsets/pitches inside the kernels. An example is seen in the [3] when the program tried to access the shared memory, however, it gives the error because of address is out of bounds. The code snippet in Figure shows that when executing the code works correctly, however, if we remove the keyword @inbounds in lines 5 and 7, the program gives an illegal memory access error at the address location 0xe7000004 is out of bounds. @inbounds macro tell the compiler to skip bounds checking within the given block. However, if the compiler checks for bounds the kernel memory address the issue of illegal memory access.



Another example is seen in [4] when the program consists of undefined behavior in the way Triton handled work from CUDA contexts lead to an illegal memory address error.

1. *Memory Copy Issues:* GPU programming issues related to this category mostly occur due to the inconsistent memory transfers between host-to-device, device-to-hose, and device-to-device. An example is seen in [7] when CuArrays.copyto!(B,A) fails to copy from a device to a host array vice versa. When both arrays are of different types, such as Float32 vs Float64, the memory copy fails due to a type mismatch.
2. *Memory Usage Issue:* GPU programming issues related to this category mostly occur due to the incorrect memory usage in the device. An example is seen in [9] when AMD and intel GPUs behaves differently when training a classifier model. Due to the small VRAM memory, it does not produce different results when allocating a large device memory. Another example is seen in [10] when the matrix size does not exactly match the size of tiles. The code snippet in figure shows that when the assigning the dimensions of grid, it is off by -1 that causes the problem due to the incorrect memory usage in the device.
3. *Environmental Issues:* These GPU programming tests cause issues due to the differences in the underlying drivers, and versions used to run the tests. The Nvidia drivers can vary from GPU program environments that it is designed to work with. Also, version issues arise due to incompatible CUDA versions used to test under different TensorFlow, pytorch versions. We found that these issues can also further divided in to three subcategories.
4. *Driver Issues:* GPU program bugs related to this category mostly occur due to the use of old drivers, and incompatible drivers. An example is seen in [11] when the program causes a page allocation failure in the Nvidia driver which makes the graphical interface to completely freeze but letting everything else to work such as sound, and keyboard responses.   
   Another example is seen in [12] when the CUDA driver version is insufficient for CUDA runtime version. The bug has occurred due to the incompatibility between the NVidia Drivers and CUDA versions.
5. *Version issues:* GPU program bugs related to this category mostly occur because of version mismatch between CUDA, TensorFlow, and pytorch. Also, some programs are designed to run in specific CUDA versions that is not compatible with other versions. An example is seen in [13] when the program did not work with CUDA 10.1 runtime version, however the CUDA 10.2 runtime version works well.   
   Another example is seen in [14] when the program gives a Runtime CUDA error due to an invalid device function. The reason for this problem is different CUDA versions in everywhere. For example, the MMCV is compiled with CUDA 9.2, PyTorch environment is built with CUDA 10.1, and the current CUDA runtime and NVCC is 10.2. A correct CUDA version that supports all other environments could fix such bugs.
6. *Device or Environmental Variable Issue:* GPU program bugs related to this category occurs due to incorrect configurations in CUDA visible devices in the CUDA driver. An example is seen in [15] when the program error due to incorrect setting of CUDA\_VISIBLE\_DEVICES that is used by the CUDA driver to decide which devices should be visible to CUDA.
7. *Synchronization Issues:* These GPU program tests cause issues because of synchronization issues between P2P copies, multiple threads sharing the same GPU, CUDA stream pipeline does not work, and implementation of asynchronous GPU transfers. An example is seen in [17] when all the cudf processing uses a single stream which blocks all other streams from sharing the GPU. The plugin uses a semaphore to prevent too many processes on the GPU at once.   
   Another example is seen in [18] when the program gives wrong results when CUDA stream is created using CU\_STREAM\_NON\_BLOCKING which specifies that the work running in the created stream may run concurrently with the work in the default stream. In the example seen in [19], another issue occurs with CUDA streams leading to invalid results when copying back the matrix from the GPU to main memory the stream is not synchronized.
8. *Compiler related issues:* These GPU program tests cause issues due to the erroneous nature in the compilers and compiler related nature. Make files can cause bugs due to incomplete configurations. Also, CUDA programs compile with NVCC compilers which the programs run with unsupported GPU architecture. Also, we categorize the precompile failure issues in GPU programs under this section.
9. *Makefile issues:* GPU program bugs related to this category occur due to incomplete or configurations in the MakeFiles that is used to compile the programs. An example is seen in [20] when it produces a CMake Error where the CUDA toolset cannot be found. The older versions of CUDA can cause this issue when the CMake cannot find the right CUDA version. However, reinstalling the CUDA version can fix the issue.
10. *NVCC issues:* CUDA programs are compiled using the NVCC compiler. However, the GPU program bugs related to this category mostly occur due to unsupported GPU architecture or undefined GPU architecture. An example is seen in [21] when an NVCC fatal error has encountered when the unsupported GPU architecture ‘compute\_80’. When the CUDA toolkit does not support compute\_80 this error occurs in the NVCC compiler.
11. *Compile Failures:* GPU program bugs related to this category occurs due to pre-compile or compiler failures. Some programs cause bugs when the program is compiled only in the Host (CPU) and cannot find the GPU during the compilation. Sometimes version mismatches can cause compile failures in the programs. An example is seen in [23] where the program encounters with a Runtime Error when it is not compiled with GPU support. Not setting CUDA\_HOME environment variable and version of CUDA can cause such bugs.
12. *Floating Point issues:* GPU architectures support single precision and double precision floating point calculations. However, these bugs can lead to floating point exceptions in the GPU perform unexpected results. These programming bugs occur due to floating point exceptions or when fp16 uses high memory than fp32 when using mixed precision architectures. An example is seen in [25] where the program works correctly in the CPU, however, with the GPU enabled it leads to a floating-point exception. The program is tested under Nvidia Tesla k80 device, but the K80 is too old for the program. Updating to a newer GPU with compatibility later than 6.0 can fix the issue.
13. *API functions / flagging issue:* CUDA provides APIs to specify thread level parallelism and device specific operations such as memory allocations and data transfers between host and device. However, the GPU program bugs may occur due to incorrect use of these functions or flagging issues. An example is seen in [27] when the CuPy program tries to use free\_all\_blocks(). According to the CuPy documents, free\_all\_blocks() has no arguments. However, cp.cuda.MemoryPool.free\_all\_blocks() in the program gives an error specifying that ‘cupy.cuda.memory.MemoryPool’ objects need an argument. The reason behind this issue is that free\_all\_blocks is an instance method, not a class method. Therefore, if CuPy is used through a Chainer, the memory pool instance is chainer.cuda.memory\_pool.free\_all\_blocks(). If a stable version of CuPy without Chainer, memory pool is not used unless the code is explicitly setting the memory pool though cupy.cuda.memory.set\_allocator. Therefore, while using CuPy from the master branch, the memory pool is cupy.\_default\_memory\_pool.
14. *Library Issues:* GPU accelerated libraries are used to make coding much more efficient and easier. However, we find that some bugs occur due to the wrong implementation or initialization of these libraries. We found four different libraries that consist of bugs when try to use them. We categorize them into four categories.
15. *Math Library issues:* CUDA programs support math.h library which can be used to call math functions to perferom necessary computations. GPU program bugs in this category occur when the CUDA accelerators handles the edge cases such as positive, negative infinity and Nan in math functions such as floor (), ceiling (), round () in the kernel. An example is seen in [24] when CUDA. Pow() fails when using float64 type values under Julia GPU computing. However, the program works when using integer power. The code snippet in the figure shows that, in the function the line 3 where the program tries to find the power of 1.5 giving an error saying cannot select f64 = fpow. However, replacing ^1.5 with ^Float32(1.5) still causes the program failure. On the other hand, the code snippet in the figure shows that the program works fine when finding the powers of integer values. It also specifies that using CUDA.pow(x,Float64(Int(y))) can fix the problem.
16. *cuRand issues*: cuRand is a library that is used to generate random numbers using CUDA. The bugs occur due to inconsistent seeding issue. An example is seen in [16] when the cuRand.seed!() doesn’t seem to perform or give results like Random.seed!() does. When the program is tested on two GPUs, both generated same random values every time indicating that cuRand.seed!() did not reseed independently.
17. *NCCL issue*: NVIDIA Collective Communication Library or NCCL is a library that implements multi-GPU and multi-node communication primitives. The library consists of APIs defined by MPI. The bugs related to this category occurs when P2P transactions fails or NCCL commands fail to during execution. When using multi-GPUs the programs fail because operations try to run using different streams and communicators and implementing wrong configurations. An example is seen in [32] when the distributed data parallel hangs when using NCCL. Both ranks used were able to load the first batch and then rank 1 got stuck and failed to advance to loading the second batch. The reason for this issue is that the subsequent subprocesses try to allocate memory from GPU 0. Therefore, assigning the rank that can access the GPU is better.
18. *Thrust issue*: Thrust is a C++ template library for CUDA which helps to implement parallel applications with minimal programming efforts. It consists of a collection of data parallel primitives to build complex algorithms. The bugs related to this category can arise due to failure in synchronization of different STL algorithms while using different threads. An example is seen in [33] when the merge sort fails to synchronize even though the program is run on separate CUDA streams with separate algorithm instances. The thrust occurs issues when working with multiple CPU threads.
19. *Visual or Graphic issues:* GPU programming is widely used is creating new video games or graphical related work. However, programs can cause issues related to graphics. An example is seen in [30] when the user interface is having issues with blue and red tint to the screen which did not occur in the previous builds. The issue has been specific to the GPU that can be fixed by changing to a stretched mode and changing the GPU’s UI scaling. Another example is seen in [31] when the program breaks the GPU. The offset and calculations and the monitor bounds are the reason for such visual bugs.
20. *Results*

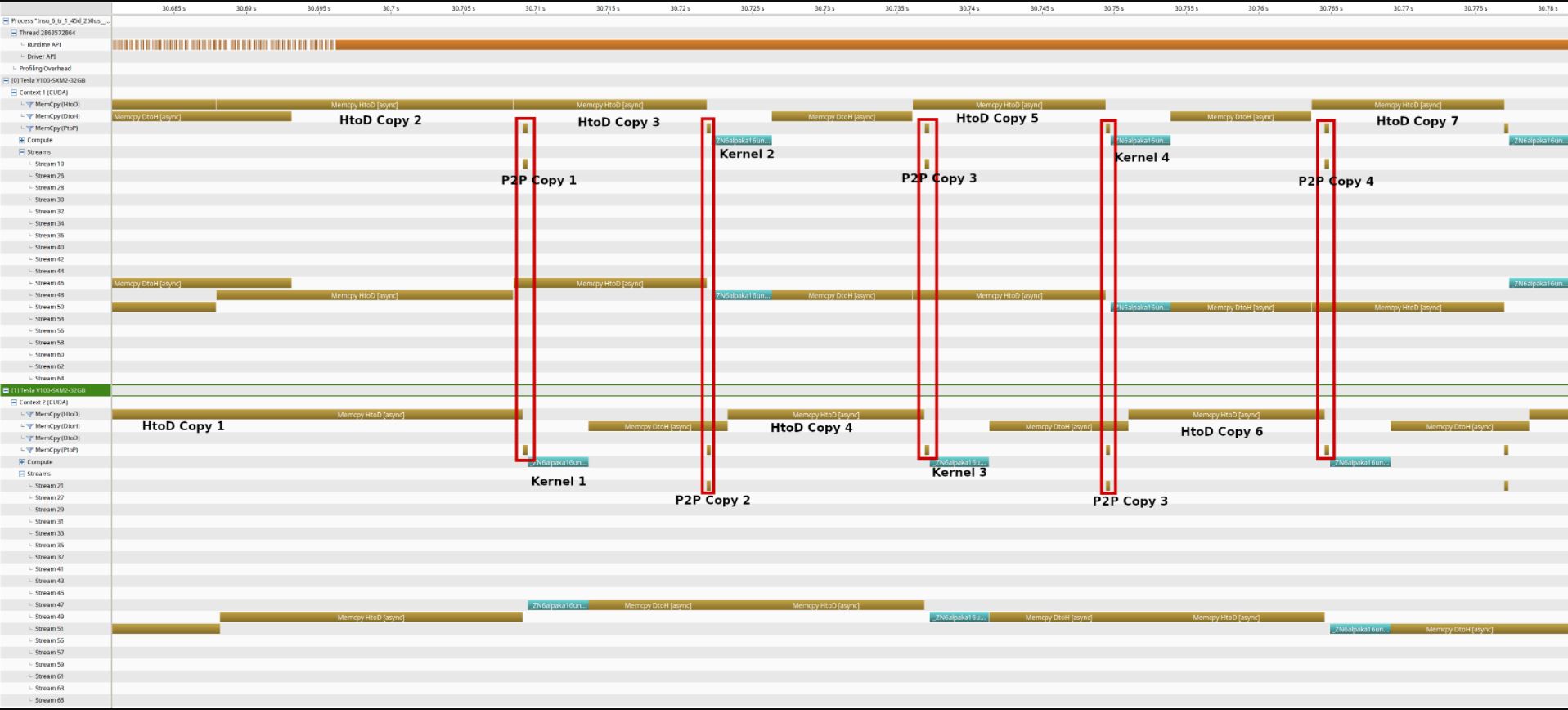
From these samples, we were able to find characteristics that are particular to GPU programming bugs. The most predominant root cause for these issues involved memory related problems such as, out of memory, illegal memory access, memory usage issues, and memory copy issues. Even though these issues are similar to each other, we categorize them in a way so that it is easy to understand the issues related to each category. Many of these issues were resolved by changing the memory size such as reducing the block sizes, reducing the input sizes, and using small batch sizes. We found that the root cause of the buggy behavior could present a challenge to find and properly fix, with some issues spanning over months to fix. Other root causes include Environment/ version, synchronizations, compiler, API function, floating point, libraries, and visual issues. Environment/ Version specific behaviors occur when GPU programs are supported to run on different versions, drivers, devices and environment settings. Synchronization problems are occurred due asynchronous memory transfers, P2P copies, and stream pipeline doesn’t work properly. Compiler related bugs are occurred due to incorrect configurations in NVCC, CMAKE files, and compile errors. GPU programs use API functions to implement kernel codes. Inconsistency, wrong values passed to these functions can cause API issues. Floating point issues are occurred when trying to use mixed precisions in the program. Also, GPU computing supports accelerated libraries such as maths, cuRand, NCCL, and thrust. These libraries can cause errors due to incorrect configurations or erroneous use of the library commands. Visual or graphics bugs occur due to incorrect UI scaling.

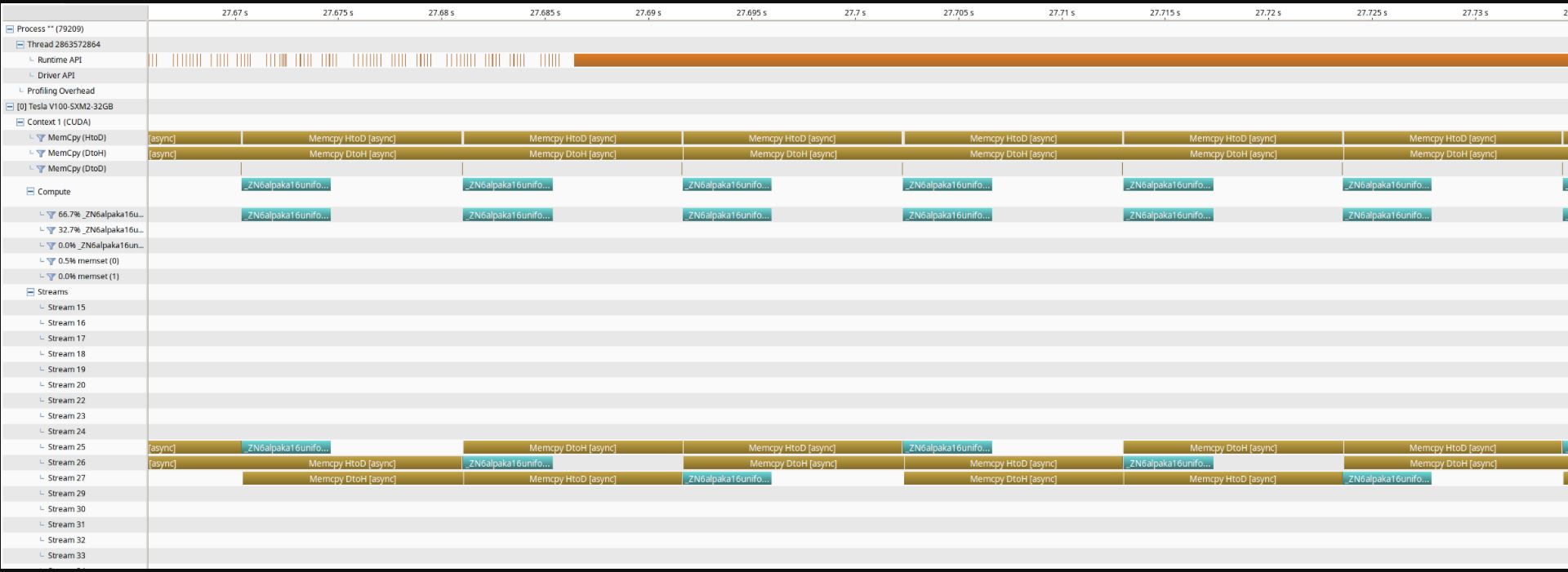
V. MANIFESTATION

Reproducing bugs is a challenging task due to the non-deterministic behavior. If developers provide details on how the bugs were initially encountered and subsequently reproduced, this information provides possible strategies to apply for similar cases. We explore the strategies used by developers to manifest the underlying buggy behavior and construct categories for similar manifestation actions taken. Our categories are summarized in Table

TABLE V: Summary of Manifestation of Categories.

|  |  |
| --- | --- |
| Manifestation Category | Total |
| Unspecified | 81 |
| Specify Memory limitations | 46 |
| Force environmental conditions and configurations | 21 |
| Provide Code snippets | 41 |
| Check for concurrency | 10 |
| Specify Problematic Platform and version | 42 |

1. *Specify Memory Limitations:* Some tests can manifest using memory limitations such as batch sizes, amount of memory used, accessing large blocks or increasing the number of blocks and changing input sizes. It helps to manifest the inconsistencies between memory errors occur in the GPUs. An example is seen in [M7] when the simulator requires more memory due to the increase of number of active blocks and there isn’t enough memory left. Nvidia System Management Interface (smi) that facilitates in monitoring and management capabilities of GPU devices is a tool that is used to check status on GPU memories in some tests.
2. *Force Environmental conditions and configurations:* The GPU bugs that manifest only when run on a specific platform or under certain environment setting can be manifested by forcibly setting these conditions such as environmental variables, device IDs and configurations. An example is seen in [f3] when the program cannot initialize multi-GPU programming and it is manifested that the by changing the environmental variable by setting CUDA\_VISIBLE\_DEVICES.   
   An example of manifesting bugs using this category is seen in [M2] where the CMake file used to compile the GPU program has stuck in an infinite loop at configuration if NVCC is not found. The error log has helped to configure that the issue is related to the NVCC in the path. The misconfiguration on the machine and removing the link between ccache and NVCC path solved the problem.
3. *Provide Code snippet:* Among the bug reports, we find that some reports include code snippets. The code snippets of these GPU program bugs make reproducing of the bugs more reliable. An example is seen in [M5] when a simple implementation of NCCL all\_reduce fails the program to complete. However, by changing the torch.cuda.set\_device(0) to torch.cuda.set\_device(rank) because in the code processes are using the same GPU and trying an all\_reduce for two tensors on the same GPU. By the changing the line of code, it helps to fix the issue.
4. *Checking for concurrency:* Some tests are reported to manifest the concurrency between memory transfers, streams, and between CUDA APIs. A tool called visual profilers are used to manifest the concurrency. Visual Profilers are used as a profiling tool that helps to receive a feedback for specially in CUDA C/C++ applications. The profilers help to identify potential performance bottleneck issues, viewing CUDA activities on both CPU an GPU memory transfers, CUDA launches, and CUDA API calls. An example is seen in [M1] when the P2P copies have issues with synchronization. Visual Profilers are used to manifest this bug when copying data from host to GPU and copying data from another GPU, both copy operations block each other resulting an inefficient schedule. The snippet in the Figure shows that HtoD Copy 2 operation seems to block P2P Copy 1 from device 0 to 1. However, all P2P copy operations could start right after the previous kernel theoretically.  
     
     
    On the other hand, while using one GPU, these P2P copy operations are just a memcpy on the device memory resulting better schedule. Therefore, the one GPU implementation outperforms the two GPU solution. The snippet in the Figure provides an overview of one GPU implementation.



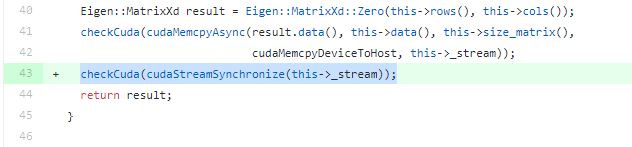
1. *Specify Problematic Platform or version:* Some tests are reported to only manifest on a specific environment or platform that the program works correctly. In this case, the author of the report specifies the problematic platform or environment version to reproduce the bugs. An example is seen [M4] when the program gives errors during the initialization of the GPUs. However, the report specifies the CUDA version used is 11.1.1-1. To fix the issue, it is said that the system should have CUDA 11.0.   
   Another example is seen in [M6] when the program cannot build TensorFlow 2.3 from source on with CUDA 11.1. However, TensorFlow version 2.3 is compatible with CUDA 10.1 and if the program needs to use CUDA 11.1, it should use TensorFlow version 2.4rc.

|  |  |  |
| --- | --- | --- |
|  | Findings | Implications |
| 1. | Of the observed GPU program tests collected, 78 tests of the 241 (32.4%) datasets are caused by memory issues | This group represents a significant portion of dataset collected and highlights the need to take this root causes into consideration when writing GPU programs. |
| 2. | Environmental issues are happening more frequently. (21.7%) | It is caused by different version mismatches between GPU programs. Some GPU programs use TensorFlow, PyTorch libraries and these GPU program versions should match to TF and Pytorch versions. Also, GPU drivers, and setting correct environmental variables are important in this category. |
| 3. | Synchronization issues are occurred in some tests (15.5%) | Synchronization issues occur when using asynchronous memory transfers, P2P copies, failures in multi thread access, CUDA streams does not work properly. |
| 4. | The most common fixing strategy for these GPU tests is by changing the API functions. (34%) | By trying to correct the parameters, using the necessary API functions, correctly implementing these API functions can fix majority of GPU program bugs. |
| 5. | Changing versions is a fixing strategy used in issues related to Environments. (22.8%) | Trying to check the correct versions such as CUDA, driver, library, TensorFlow, pytorch versions should be first considered strategies for developers when fixing bugs. |
| 6. | Fix memory can be done by different strategies | Changing the memory size, and input sizes can be used to fix the issues. Changing batch sizes can be used to fix memory issues when using GPU programs with TensorFlow. |
|  |  |  |

VI. FIXING STRATERGY

In this section, we examine the fixes of the GPU program bugs. We identify common fixing patterns and group them into categories. Through comparative analysis of root causes and fixing strategies, we find that most memory issues are fixed by changing the memory sizes, batch sizes and input sizes accordingly. The issues caused by environmental issues such as driver, version issues can be solved by changing versions by upgrading or downgrading. Majority of root causes such as synchronization issues, library issues, API function issues, graphic issues can be solved by altering program, especially by fixing API functions related to the buggy nature. Table summarizes the categories and distribution of fixing strategies.

|  |  |  |
| --- | --- | --- |
| Categories | Subcategories | Total |
| Fix Memory | Change batch size  Change memory size  use pinned memory pool | 17 32 3 |
| Dependency | Change environment reinstall packages correct visible devices and validate GPUs | 55 1 14 |
| Alter Program | Fix API functions correct Types  Fix seeds UI scaling | 82 3 2 4 |
| Add GPU support and configurations | Add GPU support and correct configurations | 18 |
| Fix MakeFiles | Change MakeFiles Correct NVCC path | 8 2 |

1. Fix Memory
2. Change Batch sizes: Some tests are resolved the bugs by fixing the batch sizes used in the program. Reducing the batch sizes to overcome out of memory issue has been a technique used by the authors to fix the program. An example can be seen in [f1] when the program leads to an out of memory issue when the program is trained using a batch\_size = 32. However, when reduce the batch size the error has been fixed. When the batch size is increased the program gives memory errors and cannot determine an exact size to avoid such errors.
3. Change memory size: When the program tries to access large number of blocks or pass larger memory transfers between host to device, there can be issues with memory. Therefore, changing the size of the memory can fix such issues. An example can be seen in [F2] when the program gives and CUDA illegal memory access issue because the device needs more memory due to increase number of active blocks and there are not sufficient blocks left. To fix this issue, they have suggested to resize the memory size.
4. Use pinned memory pool: Another fixing strategy that is been proposed is to use pinned memory pool to transfer data between host and device.
5. Dependency
6. Change environment: GPU programs work well on a certain version. However, when using different versions, drivers, and devices most programs will cause issues. The best way to fix such issues is by changing to the correct versions. An example can be seen in [12] when the CUDA driver version is insufficient for CUDA runtime version. The error is occurred due to the incompatibility between NVIDIA driver versions. The possible fix for this issue is using CUDA 10.1 requires version >= 418.39. Another example is seen in [14] when the program gives a runtime error. The program uses incompatible versions of CUDA, PyTorch with current runtime and NVCC versions. It is suggested to use PyTorch with CUDA 10.2 version and it fixes the issue.
7. Reinstall packages: It is suggested that fresh installation of packages can fix issues related to environment.
8. Correct visible devices and validate GPUs: Environmental variables such as CUDA\_VISIBLE\_DEVICES is use by the CUDA driver to decide what devices should be visible to CUDA. By specifying it correctly, it can fix issues related to environment variable in GPU programs. An example is seen in [f3] when the multi-GPU initialization fails. However, by setting CUDA\_VISIBLE\_DEVICES = 0,1,2 fixes the issue.
9. Alter the Program
10. Fix API functions: Some tests are resolved by fixing the usage of API function. After correcting the API functions, the tests behaved as expected. An example can be seen in [f4] when the program gives invalid results because the stream is not synchronized when copying the matrix from device to host. By correcting the API function, it solved the problem. The code snippet in the figure shows that by correcting the CUDA stream synchronize function fix the issue.
11. Fixing seeds: By changing the seeds in cuRand libraries can fix the issues related to random seeding.
12. UI scaling: Visual bugs can be corrected by scaling the UI correctly according to correct dimensions. Boundary fixing is used fix the issues.
13. Add GPU support and correct configurations: Some tests fail due to errors such as “no GPU found” and such errors can cause misleading errors. By adding GPU support and correct initialization support these errors can be fixed. An example is seen in [f5] when the CUDA errors cannot be immediately identified. By adding CUDA support in the program, the problem can be fixed.

1. Fix Makefiles
2. Change Makefiles: Compiler related issued can fixed by changing the makefiles and CMake files by adding correct versions, and paths.
3. Correct NVCC paths: By adding correct NVCC paths, it is possible to make the programs work correctly.

VII. DISCUSSION AND IMPLICATIONS

We investigate our collected GPU program tests to identify the relationships between the root causes, manifestation strategies, and fixing strategies defined in sections IV, V, VI, respectively.

Through our inspection, we can identify relationships between the underlying root causes in issues and how the issue was fixed. These relationships are presented in Figure

The goal of our study on GPU Program bugs is to gain insights on different GPU programming issues and fixing approaches, so we analyze our dataset to identify correlations between manifestation strategies and root causes. However, we find that no strong correlations between these two groups exist in the dataset. Similarly, we could not identify not identify strong correlations between manifestation strategy and fixing strategy. This leaves the question of detection strategies for GPU program bugs left open for future work to address. Our results do support relationships between root causes and fixing strategies. If the root cause of a GPU program bug is known, the relationships we draw in Figure can be used to select an appropriate fixing strategy.

Preliminary design ideas can be made for some fixing strategies we identify in section VI. For *Changing Memory Size* fixing strategy, a possible implementation is to check the statements that allocate memory in the kernel functions, such as, input size, block size, amount of memory transfers. Using the relationships found in Figure, this approach can be used to fix issues caused by *Illegal Memory Access* *issues*, *Memory Copy issues*, and *Memory Usage Issues*. For the *Change Environment* fixing strategy, an approach to repair would be identify the environment that program works and the environment that the system uses such as, drivers, versions, and the device numbers that is working on the kernel. This fixing strategy is helpful to fix issues related *Memory issues, Drivers issues, Version Issues, Environment Variable issues, synchronization issues,* and *compiler Issues*. The *Fix API Functions* fixing strategy can be implemented by checking for API functions correctness. These issues are related to programming issues where incorrect use of API functions can lead to buggy behavior in the program. This approach can be used to address issues related *to Memory issues, Synchronization Issues, API Function flagging Issues, Floating Point Issues,* and *Library related Issues*.

VIII. THREATS TO VALIDITY.

The results of our study are subject to several threats, including the representativeness of the projects inspected, the correctness of the methodology used, and the generalizability of our observations.

Regarding the representativeness of the projects in our dataset, we focused on repositories associated with GPU Programming and we restrict the repositories to focus on repositories that impact real applications. We searched through GitHub database with strict and clear condition settings to ensure that the samples we obtain are targeted and representative.

In our correctness of the methodologies used, we collect all available commits on GitHub from the repositories related to GPU programming. We also leveraged the GitHub Archive repository to find all issues related to GPU programming in the year 2020. We filter our relevant commits and issues using keywords and then manually inspect the remaining commits and issues in order to verify the relevance to GPU programming bugs.

Regarding the generalizability of the implications made, we selected the GPU programming samples from the actual projects used in the wild. We limit numeric implications only to the dataset collected, and focus on qualitative implications made on the features of the test samples.

IX. RELATED WORK

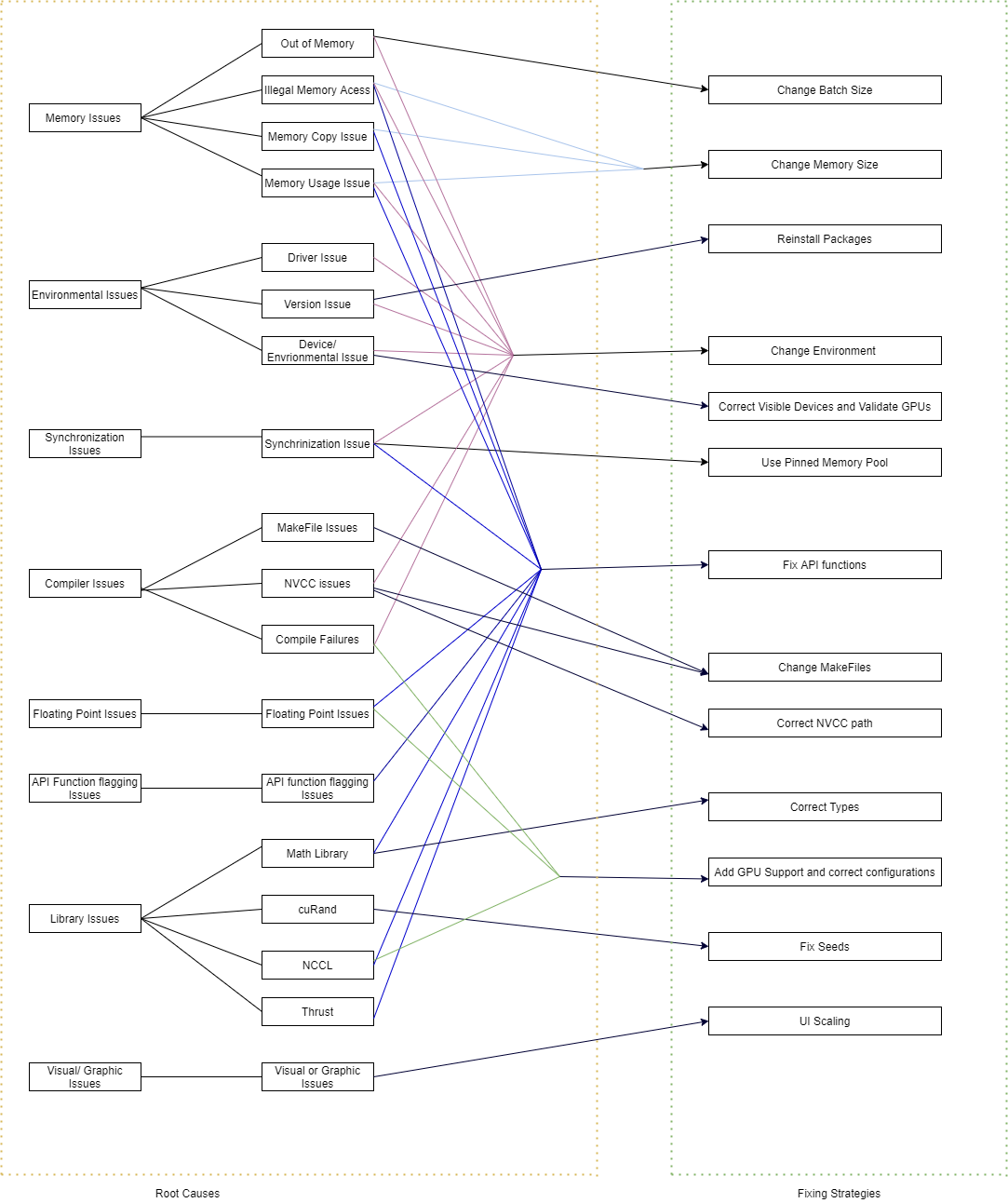
Empirical Studies on GPU program bugs. There have been several prior studies analyzing the bugs related to GPU programming [R10, R8, R6]. For example, in Yang et al. [R8] an empirical study was conducted on Fixing Performance bugs on Open-source GPGPU system. In Tiwari et al. [R10] understanding GPU errors on large scale HPC systems and the implications for system design and operations. In Li et al. [R6], undefined behaviors in CUDA C programs were detected.

Detecting GPU programming bugs: In Wu et al. [R9], developed a framework called Simulee to detect CUDA synchronization bugs via Memory-Access Modeling. In Sorensen et al. [R7] presents a systematic design of a testing environment to reveal errors in GPU applications that arise due to weak memory effects. In Islam et al. [R4], proposes Bugaroo to expose memory model bugs in any arbitrary GPU program. In Boyer et al.[R2], presents an automated dynamic analysis technique for finding bugs related to race conditions, and memory bank conflicts in CUDA programs.

In Wang et.al [R5], propose CUDAsmith, a fuzzing framework for CUDA compilers that can randomly generate deterministic and valid CUDA kernel code with several different strategies. Also, in Yang et al. [R3] addresses particular challenges facing the real-time community when utilizing CUDA-enabled GPU programs for autonomous applications, and best practices for applying real-time safety-critical principles.

X. CONCLUSION

This paper performs a study on bugs arising in GPU programming. We investigated 241 tests collected from GitHub repositories. The test samples are analyzed to identify the typical root causes of the buggy nature, the manifestation strategies used to report and reproduce the bugs, and the common fixing strategies applied to these tests to reduce the buggy behavior. Through our analysis, we present findings on the prevalence of certain root causes, the differences that root causes appear, and the differences in the fixing strategies applied. We believe our analysis can provide guidance towards developing effective detection and prevention techniques specifically geared towards GPU programming bugs.



Memory   
 Out of memory   
 Illegal Memory Access/ allocation   
 Memory Copies  
 Memory Usage

Environment   
 Driver issues   
 Version Issues   
 Devices Issue/ Environmental Variable

Synchronization (Streams, threads)

Compiler   
 Make Files   
 NVCC compiler   
 Compile   
  
  
Floating point issues   
API function/ Flagging Error

Library   
 Math library  
 NCCL  
 Thrust   
 CuRand   
   
Visual/ Graphics issues

Other

Fixes   
Reduce batch sizes, block size, increase memory, limit GPU memory,

Change GPU type, correct package installation, update drivers, update versions,

Use pinned memory, improve multi-stream/ GPU

Matching NVCC versions, correct NVCC path, build make files correctly,

Fix code, correct dimensions, correct NCCL initialization, boundary checking,

Memory   
 Out of memory

1. "https://github.com/oreilly-japan/deep-learning-from-scratch-2/issues/16" (fix)
2. "https://github.com/PlasmaControl/DESC/issues/1"

Illegal Memory Access/ allocation

1. "https://github.com/JuliaGPU/CUDA.jl/issues/558"
2. "https://github.com/ptillet/torch-blocksparse/issues/15"
3. "https://github.com/microsoft/onnxruntime/issues/5555"

Open Issue but solve the problem:  
0pen1: "https://github.com/cupy/cupy/issues/3452"

Most common mistake for that is wrong index computation for arrays or wrong array offsets/pitches etc. inside the kernels.  
You'd get other errors if you'd run out of resources, e.g. too many streams.

Memory Copies

1. "https://github.com/rapidsai/dask-cuda/issues/438"
2. "https://github.com/JuliaGPU/CUDA.jl/issues/105"

Memory Usage   
  
8. "https://github.com/uber-research/LaneGCN/issues/2"  
9. "https://github.com/sowson/darknet/issues/38"

10. "https://github.com/codezonediitj/adaboost/issues/8"

Environment   
 Driver issues

11. "https://github.com/yshui/picom/issues/537"

12. "https://github.com/rsanchezgarc/deepEMhancer/issues/4"

Version Issues

13. "https://github.com/tmcdonell/cuda/issues/66"

14. "https://github.com/open-mmlab/mmdetection/issues/4012"

Devices Issue/ Environmental Variable

15. "https://github.com/PyTorchLightning/pytorch-lightning/issues/2420"

Synchronization   
17. "https://github.com/NVIDIA/spark-rapids/issues/15"

18. "https://github.com/m4rs-mt/ILGPU/issues/222"

19. "https://github.com/StanfordLegion/legion/issues/936"

Compiler   
 Make Files

20. "https://github.com/NVIDIA/gvdb-voxels/issues/87"

NVCC compiler

21. "https://github.com/NVIDIA/cuda-samples/issues/44"

Compile

23. "https://github.com/ying09/TextFuseNet/issues/27"

Floating point   
 25. "https://github.com/limbo018/DREAMPlace/issues/21"

API function/ Flagging Error   
26. "https://github.com/halide/Halide/issues/5443"  
27. "https://github.com/cupy/cupy/issues/578"

Library   
  
 Math library  
 24. "https://github.com/JuliaGPU/CUDA.jl/issues/71"  
 cuRand   
16. "https://github.com/JuliaGPU/CuArrays.jl/issues/589"

NCCL

32. "https://github.com/pytorch/pytorch/issues/46259"

Thrust   
33. "https://github.com/opencv/opencv/issues/18051"

Visual/ Graphics issues

30. "https://github.com/runelite/runelite/issues/12777"

31. "https://github.com/wiremod/wire/issues/1998"

Manifestation   
Concurrency  
M1. "https://github.com/alpaka-group/alpaka/issues/1216"

Force environment conditions  
M2. "https://github.com/apache/incubator-mxnet/issues/17761"

Platform/ Environment

M4. "https://github.com/uncomplicate/deep-diamond/issues/3"

M6. "https://github.com/tensorflow/tensorflow/issues/45001"

"https://github.com/colmap/colmap/issues/1035"

Code

M5. "https://github.com/pytorch/pytorch/issues/43345"

Memory   
M7. "https://github.com/penn-graphics-research/claymore/issues/3"

Fixes   
Change Batch size   
F1. "https://github.com/tensorflow/tensorflow/issues/45081"

Change memory size   
F2. "https://github.com/penn-graphics-research/claymore/issues/3"

Change environment- use above root causes

Correct visible devices   
F3. "https://github.com/LaurentMazare/tch-rs/issues/153"

FIX API   
f4. "https://github.com/NLESC-JCER/EigenCuda/issues/22"

GPU support  
F5. "https://github.com/StanfordLegion/legion/issues/911"

RELATED WORKS.

# R1. An Empirical Study on TensorFlow Program Bugs∗ R2. Automated Dynamic Analysis of CUDA Programs R3. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems R4. Bugaroo: Exposing Memory Model Bugs in Many-core Systems R5. CUDAsmith: A Fuzzer for CUDA Compilers R6. Detecting Undefined Behaviors in CUDA C R7. Exposing Errors Related to Weak Memory in GPU Applications R8. Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs R9. Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling R10. Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation R11. Hydra: Efficient Detection of Multiple Concurrency Bugs on Fused CPU-GPU Architecture

# R12. GPUburn: A System to Test and Mitigate GPU Hardware Failures