Massively Parallel, Highly Efficient, but What About the Test Suite Quality?
Applying Mutation Testing to GPU Programs

Qianqian Zhu
Delft University of Technology
Email: qianqian.zhu@tudelft.nl

Andy Zaidman
Delft University of Technology
Email: a.e.zaidman@tudelft.nl

Abstract—Thanks to rapid advances in programmability and performance, GPUs have been widely applied in High-Performance Computing (HPC) and safety-critical domains. As such, *quality assurance* of GPU applications has gained increasing attention. This brings us to mutation testing, a fault-based testing technique that assesses the test suite quality by systematically introducing small artificial faults. It has been shown to perform well in exposing faults. In this paper, we investigate whether GPU programming can benefit from mutation testing. In addition to conventional mutation operators, we propose nine GPU-specific mutation operators based on the core syntax differences between CPU and GPU programming. We conduct a preliminary study on six CUDA systems. The results show that mutation testing can effectively evaluate the test quality of GPU programs: conventional mutation operators can guide the engineers to write simple direct tests, while GPU-specific mutation operators can lead to more intricate test cases which are better at revealing GPU-specific weaknesses.

I. INTRODUCTION

A graphics processing unit (GPU) is a single-chip processor originally used to boost the performance of video and graphics. The recent development of massive parallelism and energy efficiency and the ease of programming using the CUDA [1] and OpenCL [2] programming models have made GPUs attractive for High-Performance Computing (HPC), which requires compute-intensive, highly parallel computation [3], [4]. Moreover, GPUs are increasingly used in some safety-critical domains, such as medical science [5] and automotive [6]. In both HPC and safety-critical domains, *quality assurance* of GPU applications is an important issue [7], [8].

However, it is not easy to write a correct GPU program [9]. The essential elements of GPU programs are kernels, which are functions executed on GPU cores. To efficiently schedule instances of kernels on a GPU platform, a programmer needs to face the challenges in *computation* and *memory access* that do not appear in Central Processing Unit (CPU) programs.

Regarding computation, a programmer tends to spend less effort on thread management on the CPU platform than on the GPU platform [9]. To ensure high execution efficiency, a CPU usually supports far fewer threads running in parallel than a GPU of the same generation. Since the operating system needs to spend extra time on thread scheduling when the number of threads triggered by an application exceeds the total number of cores on the CPU. As a result, a CPU programmer may focus on the correctness and efficiency of a single thread, and leave the thread collaboration issues to libraries (e.g., OpenMP) and third-party tools. In contrast, managing threads on GPUs is challenging and rarely automated as a GPU often contains thousands of cores. Development toolkits as CUDA require programmers to manage the threads explicitly.

With regard to memory access, the hardware-level facilities for memory, such as the memory hierarchy, are typically transparent to programmers for the CPU. For instance, a CPU programmer may reorder entries in an array without knowing or controlling the movement of data in between the main memory and CPU cache. In contrast, a GPU platform has a separate memory space which is isolated from the main memory of the host computer. To ensure correctness and execution efficiency, GPU programmers have to explicitly manage different types of memory including shared memory, global memory and the memory of host computer [10].

Given the increasing demand for quality assurance of GPU applications as well as the challenges in GPU programming, it becomes essential to understand to which extent a GPU program can be analysed and tested. This brings us to *mutation testing*, a fault-based testing technique that measures the test effectiveness in terms of fault detection [11], [12]. Mutation testing has been shown to perform well in exposing faults compared to other test coverage criteria [13]–[15]. Also, mutants can act as a valid substitute to real faults [16], [17].

In this paper, we aim to enable mutation testing for GPU programming to investigate if mutation testing can help in GPU program testing. To achieve this goal, we develop a mutation testing tool named MUTGPU especially for GPU applications in the CUDA programming model. Considering the differences between CPU and GPU programming, we design nine new GPU-specific mutation operators in addition to conventional mutation operators. We perform an empirical study involving six GPU projects from the CUDA SDK [18]. To steer our experimental study, we propose the following research questions :

RQ1 *How frequently can GPU-specific mutation operators be applied, and how good is the existing test suite at killing them?*

RQ2 *How effective are conventional mutation operators in evaluating the test suite of GPU programs?*

RQ3 *How effective are GPU-specific mutation operators in evaluating the test suite of GPU programs?*

RQ4 *How do GPU-specific mutation operators compare with conventional mutation operators in terms of the improvement?*

Fig. 1. Comparison of CPU and GPU [1]

II. BACKGROUND

A. GPU computing

A graphics processing unit (GPU) was originally dedicated to providing a high-performance, visually rich, interactive 3D experience [19]. With rapid advances of programmability and performance, a GPU becomes a compelling platform for computationally demanding tasks in various domains [20].

GPU computing, also known as general-purpose computing on the GPU (GPGPU), is to use a GPU as a co-processor to accelerate CPUs for general-purpose scientific and engineering computing [20]. Compared to the CPU, the GPU contains many more transistors devoted to data processing rather than data caching and flow control (as demonstrated in Figure 1) [1]. Thus, the GPU is especially well-suited for compute-intensive, highly parallel computation.

Another important characteristic of GPUs is that the programmable units (a GPU core) follow a "SPMD" programming model: single program, multiple data [20]. More specifically, the GPU processes many elements (vertices) in parallel using the same program, and each element is independent of the other elements.

So far, there are three major GPU programming models: OpenCL (Open Computing Language), from the Khronos Group [2]; CUDA, from NVIDIA [1]; and C++ AMP, from Microsoft [21]. CUDA, as a popular GPU platform and programming model, was introduced by NVIDIA in November 2006. CUDA comes with a software environment that enables developers to program in C/C++. Except for C/C++, CUDA also supports Fortran, DirectCompute, OpenACC and Python.

B. Example of GPU Programming

In this section, we are going to demonstrate the main concepts of GPU programming along with a simple program in *CUDA C*[1]. This program is to sum two vectors (*a* and *b*) into a third vector (*c*). In standard C, we can easily compute within a *for* loop shown as Listing 1. In CUDA C, we can accomplish the same addition on a GPU by introducing a *device* function. Here, we refer to the CPU and the system memory as the *host* and the GPU and its memory as the *device*. As shown in Listing 2, we add *__global__* to *sum()* (Line 1 in Listing 2) in order to notify the compiler that this function should be compiled to run on a device instead of

[1]CUDA C is standard C with some ornamentation to allow developers to specify which code should run on the GPU and which should run on the CPU [22]

```
1  void sum(int n, float *a, float *b, float *c){
2    for (int i = 0; i < n; i++){
3      c[i] = a[i] + b[i];}}
4  int main(){
5    ...
6    int N = 1<<20;
7    // Perform SUM on 1M elements
8    sum(N, a,b,c);
9    ...}
```
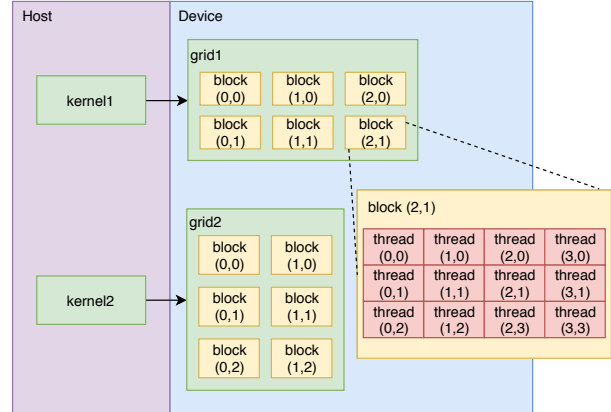
Listing 1. *sum* function in Standard C
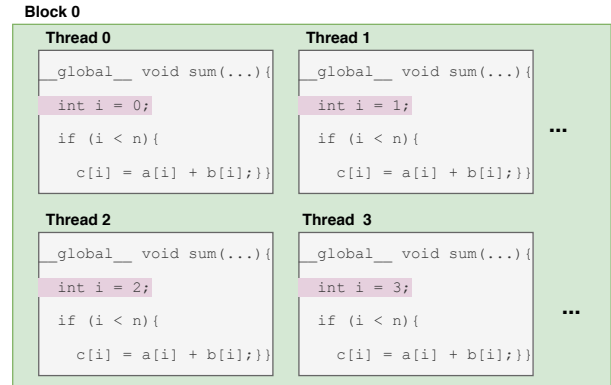

Fig. 2. CUDA programming model [1]

Fig. 3. Actual code in CUDA parallel threads

the host. Moreover, we need to determine how many parallel copies of *sum()* function (named a *kernel* meaning a function that executes on the device) to launch. A kernel is executed by a collection of thread blocks (called a *grid*), and a thread block can be further split into threads (shown in Figure 2). The statement *sum<<<4096,256>>>* (Line 13 in Listing 2) specifies to launch 1,048,576 parallel threads (4096 blocks × 256 threads per block) for function *sum*.

After we launch the kernel with 1M parallel threads, the CUDA runtime assign varying values to those threads by *blockIdx.x*blockDim.x + threadIdx.x*[2] (Line 2 in Listing 2), the first taking *0* and the last taking *n-1*. Thus, all the threads run the same instructions but with different indices in parallel. Figure 3 presents the actual code being executed in threads.

Last but not least, in order to call the kernel, we first need to load the values of the two vectors (*a* and *b*) to the device (*dev_a* and *dev_b*) by invoking *cudaMemcpy* (Line 10 and 11 in Listing 2). The function *cudaMemcpy*, similar to *memcpy*

[2]Those are built-in variables in CUDA runtime that contains the value of thread index for whichever thread is currently running the device code.

```
1  __global__ void sum(int n, float *a, float *b, float *c){
2    int i = blockIdx.x*blockDim.x + threadIdx.x;
3    if (i < n){
4      c[i] = a[i] + b[i];}}
5  int main(){
6    ...
7    int N = 1<<20;
8    int *a, *b, *c; // memory in host
9    int *dev_a, *dev_b, *dev_c; // memory in device
10   cudaMemcpy(dev_a, a, N, cudaMemcpyHostToDevice);
11   cudaMemcpy(dev_b, b, N, cudaMemcpyHostToDevice);
12   // Perform SUM on 1M elements
13   sum<<<4096,256>>>(N, dev_a, dev_b, dev_c);
14   cudaMemcpy(c, dev_c, N, cudaMemcpyDeviceToHost);
15   // examine the correct answer in c
16   ...}
```

Listing 2. *sum* function in CUDA C

```
1  void test(){
2    ...
3    int N = 1<< 25 ;
4    int *a, *b, *c; // memory in host
5    int *dev_a, *dev_b, *dev_c; // memory in device
6    cudaMemcpy(dev_a, a, N, cudaMemcpyHostToDevice);
7    cudaMemcpy(dev_b, b, N, cudaMemcpyHostToDevice);
8    sum<<<4096,256>>>(N, dev_a, dev_b, dev_c);
9    cudaMemcpy(c, dev_c, N, cudaMemcpyDeviceToHost);
10   ...}
```

Listing 3. A test case for *sum* function (Listing 2)

in standard C, controls the memory copy between the device and the host. After the execution of the device function in the GPU, we then copy the output from the device (*dev_c*) to the host (*c*) (Line 14 in Listing 2).

III. MOTIVATION

From Section II, we can see there are important differences in the programming models of the CPU and the GPU. This raises the question of whether conventional mutation operators for C/C++ are enough to represent bugs in GPU programming? We observe that a test suite mutated with conventional operators may be insufficient: the GPU code with certain issues can still easily pass the test suite.

The first example is the *memory management* in GPU. In GPU programming, we first need to specify the number of parallel processors to launch in the device. In Listing 2, we can see that 1,048,576 parallel threads are used for the kernel. In this example, the size of our testing data is 2^{20}, the exact same size as the number of parallel threads; this means that every thread can process one index of the vectors. Thereby, this test can easily pass. What if the testing data exceeds 2^{20} (such as 2^{25} shown in Listing 3)? For this test case, the specific parallel threads are not enough to iterate and compute each index individually. Therefore, the test in Listing 3 fails. This exposes a bug in the *sum* function: we have to modify the CUDA C code to allow certain threads to compute more than one index of the vectors (see in Listing 4). For Listing 2, existing mutation operators for C cannot target the problem related to parallel processor allocation in GPUs.

Another instance we observe is the *thread management* in the GPU. Different from the CPU, GPU computing involves massive parallel operations via threads; the bugs in thread management are hard to represent with conventional mutation

```
1  __global__ void sum(int n, float *a, float *b, float *c){
2    int i = blockIdx.x*blockDim.x + threadIdx.x;
3    while (i < n){
4      c[i] = a[i] + b[i];
5      i += blockDim.x * gridDim.x; }}
```

Listing 4. Modified *sum* function in CUDA C

```
1  __global__ void sum(int n, float *a, float *b, float *c){
2    int i = threadIdx.x*blockDim.x + blockIdx.x ;
3    while (i < n)){
4      c[i] = a[i] + b[i];
5      i += blockDim.x * gridDim.x; }}
```

Listing 5. *sum* function in CUDA C with indexing bugs

operators. For example, Listing 5 presents one example of indexing bugs that could occur in GPU programs.

We would also like to mention *atomic operations* in GPU programming. When developing conventional single-threaded applications, there is no need for atomic operations. However, for GPU applications which are multithreaded by default, we do need a way to perform read-modify-write without being interrupted by another thread in certain conditions, such as reduction. Atomic operation omissions are one common mistake that happens in GPU programming, e.g., Listing 6.

To sum up, we do see a necessity to investigate mutation testing specifically for GPU programming.

IV. MUTATION OPERATORS FOR GPU PROGRAMMING

A. *GPU-Specific Mutation Operators*

Mutation operators are well-defined rules to specify the syntactic changes to generate faulty versions (called *mutants*) [23]. They are the key to mutation testing, where good mutation operators lead to effective test suites while poor mutation operators generate many trivial and redundant mutants.

To design mutation operators, we usually follow two methodologies [24]: the first is based on fault models, and the other is to analyse the syntax of the language being mutated. In our study, we mainly follow the later guideline: we have defined GPU-specific mutation operators based on the core syntax differences between CPU and GPU programming (as discussed in Section III). Meanwhile, we also consider the syntactic changes in ways that programmers could make mistakes in GPU programming (the first guideline). To verify that the mutation operators we have proposed represent the common mistakes in GPU programming, we have searched for them on *StackOverflow* with the keyword "cuda" + issue name. For example, the keyword for the shared memory issue is "cuda shared memory". After searching for "cuda shared memory", we have obtained 500 search results from StackOverflow sorted by relevance. We have analysed the first 150 items. Among those, 48% are referring to bug issues.

We categorise the GPU-specific mutation operators according to the key syntactic differences between CPU and GPU programming. In the following sections, we describe the GPU-specific mutation operators we proposed by category.

1) Memory management:

```
1  __global__ void histogram(unsigned char *buffer,
2          long size, unsigned int *histo){
3  int i = blockIdx.x*blockDim.x + threadIdx.x;
4  while (i < n){
5  histo[buffer[i]] += 1; i += blockDim.x * gridDim.x;}}
```

Listing 6. *histogram* in CUDA C with atomic operation omission

a) Execution configuration: As mentioned in Section II-B, we need to specify the *execution configuration* for a kernel function. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device. To determine the number of parallel processors allocated for the kernel, many factors should be taken into consideration, for instance, the maximum size of the data and the limitation of the GPU device. For the execution configuration is unique in GPU programming, we propose the following three mutation operators to cover this aspect:

- *alloc_swap*: to replace the number of threads with the number of blocks in parallel processor allocations (and vice versa). The same bug was posted in *SO29158775* [25].

```
add<<<4096,256>>>(N, a, b, c); // original
→ add<<<256,4096>>>(N, a, b, c); // mutant
```

Listing 7. Example of *alloc_swap* mutator

- *alloc_increment*: to increase the number of parallel processors (in both threads and blocks) allocated by one

```
add<<<4096,256>>>(N, a, b, c); // original
→ add<<<4096+1,256>>>(N, a, b, c); // mutant
```

Listing 8. Example of *alloc_increment* mutator

- *alloc_decrement*: to decrease the number of parallel processors (in both threads and blocks) allocated by one

```
add<<<4096,256>>>(N, a, b, c); // original
→ add<<<4096-1,256>>>(N, a, b, c); // mutant
```

Listing 9. Example of *alloc_decrement* mutator

b) Shared memory: The shared memory in GPU programming provides a means by which threads within a block can communicate and collaborate on computations [1]. To declare the variable in shared memory, we use the *__shared__* memory space specifier in CUDA C, for instance, *__shared__ float share[64];*. The shared memory management is the main cause of *data races* and *bank conflicts*. The mutation operator *share_removal* is introduced to represent such bugs in GPU programs. There are a great deal of questions posted on StackOverflow addressing the confusion about the shared memory in GPU programming, e.g., *SO25255699* [26] and *SO9488590* [27].

- *share_removal*: to remove the shared memory space specifier in variable declarations

```
__shared__ float cache[N]; // original
→ float cache[N]; // mutant
```

Listing 10. Example of *share_removal* mutator

2) Thread management:

a) GPU indexing: GPU programming introduces a new indexing mechanism to iterate the data and threads using built-in variables such as *threadIdx.x* and *blockIdx.x* (as mentioned in Section III). While in conventional imperative programming, we use the loop statement (e.g., *for* and *while*) to iterate the data and threads. Since the indexing scheme is quite different from serial code, there are numerous questions posted on StackOverflow addressing the confusion about GPU indexing, such as *SO9859456* [28], *SO21677559* [29] and *SO33159171* [30]. Therefore, we design three mutation operators address the indexing issue:

- *gpu_index_replacement*: to replace the thread indexing variable (*threadIdx*) with the block indexing variable (*blockIdx*) and vice versa

```
int tid = blockIdx.x; // original
→ int tid = threadIdx.x; // mutant
```

Listing 11. Example of *gpu_index_replacement* mutator

- *gpu_index_increment*: to increase the indexing variables (*threadIdx* and *blockIdx*) by one

```
int tid = blockIdx.x; // original
→ int tid = blockIdx.x+1; // mutant
```

Listing 12. Example of *gpu_index_increment* mutator

- *gpu_index_decrement*: to decrease the indexing variables (*threadIdx* and *blockIdx*) by one

```
int tid = blockIdx.x; // original
→ int tid = blockIdx.x-1; // mutant
```

Listing 13. Example of *gpu_index_decrement* mutator

b) Synchronisation functions: The synchronisation function, also called *barrier*, is used to coordinate communications between threads in a specific block (e.g., *__syncthreads()* function in CUDA C). Many *data races* and *deadlocks* are caused by the incorrect barrier placement. We propose *sync_removal* to mimic the mistakes of the incorrect barrier placement. The omission or misplacement of the synchronisation function are very common in GPU code, e.g., *SO29233426* [31].

- *sync_removal*: to remove the synchronisation function call (e.g. *__syncthreads()*)

```
int i = blockDim.x/2;
while (i != 0) {
  if (cIndex < i)
    cache[cIndex] += cache[cIndex + i];
  __syncthreads();  //original
→ //__syncthreads();  //mutant
  i /= 2;}
```

Listing 14. Example of *sync_removal* mutator

3) Atomic operations: Atomic operations are unavoidable in multithreaded applications in the sense that they are guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. However, if the programmer does not pay enough attention, he or she is very likely to omit the atomic operations in GPU programming, for example, *SO14057678* [32].

- *atom_removal*: to remove the atomic functions (e.g., *atomicAdd()*) with non-atomic operations

```
while (i < size) {
 atomicAdd(&(histo[buffer[i]]),1); //original
→ histo[buffer[i]]) += 1; //mutant
 i += stride;}
```

Listing 15. Example of *atom_removal* mutator

B. Conventional Mutation Operators

CUDA C is an extension of standard C [1], therefore, conventional mutation operators for C also apply to CUDA C. Since the existing C mutation tools, such as Mull [33], cannot fully parse the grammar of CUDA C, we have to define the grammar of CUDA C first. So far, we have implemented five mutation operators which are most widely adopted in mutation testing, i.e., *conditional boundary replacement*, *negate conditional replacement*, *math replacement*, *increment replacement* and *logical replacement*.

C. GPU-specific v.s. Conventional Mutation Operators

The design principle of CUDA C/C++ is based on the traditional C/C++ syntax, which makes it easy to learn and use for developers. Although our newly proposed GPU-specific mutation operators seem to be subsumed by the existing mutation operators in terms of syntax, they are semantically different. Take *gpu_index_replacement* from GPU-specific mutation operators and *array_reference* from conventional mutation operators for example. The operator *gpu_index_replacement* replaces the thread indexing variable (*threadIdx*) with the block indexing variable (*blockIdx*). The thread/block indexing variable on the GPU is not equivalent as the array variable on the CPU. The thread/block indexing variable is used to access the parallel processors on the GPU, while *array_reference* variable is used to access the memory blocks on the CPU. One thing to notice here is that the mutants generated by GPU-specific mutation operators are totally different from the conventional operators; this means there is no overlap between those two sets of mutants.

To sum up, we have designed nine mutation operators (summarised in Table I) to replicate common errors in GPU programming. Meanwhile, we have also implemented five conventional mutation operators that can be applied to CUDA C programs (also included in Table I).

V. Tool Implementation

As mentioned earlier in Section II, there are three major GPU programming models, i.e., OpenCL, CUDA and C++ AMP. In this paper, we select CUDA as the target model to implement the aforementioned mutation operators.

To evaluate our approach, we have implemented a prototype tool (coined MUTGPU) in Python to apply mutation testing in GPU programs. Figure 4 presents an overview of the architecture of MUTGPU [35]. MUTGPU consists of two components, i.e., the mutation engine and the test executor. MUTGPU takes the program and its test suite as input. First, the mutation engine analyses the source code and marks all
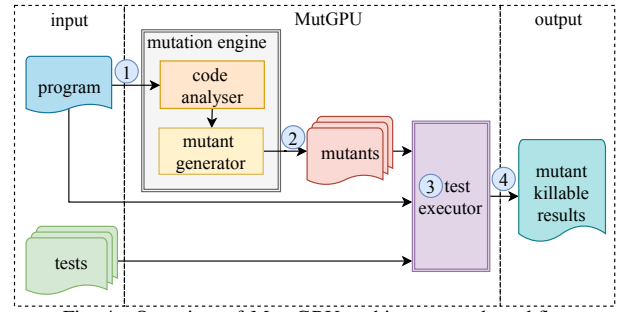

Fig. 4. Overview of MUTGPU architecture and workflow

possible mutation points, and then the mutation generator produces all the mutants according to mutation operators. After that, the program and generated mutants together with the test suite go to the test executor, where the mutation testing is performed: each mutant is executed against the test suite one by one. Finally, MUTGPU prints out the detailed mutant killable results.

The main task of the code analyser is to analyse the test dependencies and parse the source code of the program for the mutation generator. We have adopted *Pyparsing* [36] as the code analyser to parse the CUDA C code. *Pyparsing* is a pure-Python class library that constructs recursive-descent parsers with ease. The mutation generator contains all the mutation operators and the details of the mutants including the mutation location (line number) and the mutation operator type.

VI. Empirical Evaluation

To assess the efficacy of our mutation testing approach, we conducted an experimental study based on the CUDA programming model. The main purpose of this study is to investigate whether GPU programming can benefit from mutation testing, so we proposed the following research questions to steer our experimental study:

- **RQ1:** *How frequently can GPU-specific mutation operators be applied, and how good is the existing test suite at killing them?*
- **RQ2:** *How effective are conventional mutation operators in evaluating the test suite of GPU programs?*
- **RQ3:** *How effective are GPU-specific mutation operators in evaluating the test suite of GPU programs?*
- **RQ4:** *How do GPU-specific mutation operators compare with conventional mutation operators in terms of the improvement?*

A. Subject Systems

We select six GPU benchmark projects from the CUDA SDK [18]. All these benchmark projects are widely used in the research domain [37]–[39]. Table II summarises the main characteristics of the selected projects. All systems are written in CUDA C, and contain a set of test cases.

We perform the experiment on two different NVIDIA graphic cards (GeForce MX150 & GTX 960) with two releases of CUDA toolkit (9.0 & 9.1) to minimise the threat caused by errors residing in hardware and CUDA toolkits.

B. Experimental Setup

To answer **RQ1**, we investigate the mutant results for each GPU-specific mutation operator we proposed in detail. More specifically, we evaluate the *frequency* of each mutation

TABLE I
SUMMARY OF MUTATION OPERATORS

category	operator	definition
C	conditional_boundary_replacement	replace the relational operators $<, \leq, >, \geq$ with their boundary counterpart (according to PIT [34])
	increment_replacement	replace increments with decrements and vice versa
	logical_replacement	replace logical operator AND (&&) with OR (||) and vice versa.
	math_replacement	replace binary arithmetic operations with another operation (according to PIT [34])
	negate_conditional_replacement	replace the relational operators with another operation (according to PIT [34])
GPU	alloc_decrement	decrease the number of parallel processors (in both threads and blocks) allocated by one
	alloc_increment	increase the number of parallel processors (in both threads and blocks) allocated by one
	alloc_swap	replace the number of threads with the number of blocks in parallel processor allocations (and vice versa)
	atom_removal	remove the atomic functions (e.g. *atomicAdd()*) with non-atomic operations
	gpu_index_decrement	decrease the indexing variables (*threadIdx* and *blockIdx*) by one
	gpu_index_increment	increase the indexing variables (*threadIdx* and *blockIdx*) by one
	gpu_index_replacement	replace the thread indexing variable (*threadIdx*) with the block indexing one (*blockIdx*) and vice versa
	share_removal	remove the shared memory space specifier in variable declarations
	sync_removal	remove the synchronisation function call (e.g. *__syncthreads()*)

TABLE II
SUBJECT SYSTEMS

Project	File	LOC Source	Test	hCOV[2]	#Mutants C	GPU
MonteCarloMultiGPU	MonteCarlo_kernel.cu	231	359	100	71	59
	MonteCarlo_reduction.cuh	71			14	2
convolutionFFT2D	convolutionFFT2D.cu	226	509	100	40	36
	convolutionFFT2D.cuh	463			250	64
histogram	histogram64.cu	219	141	100	82	96
	Histogram256.cu	165			48	81
mergeSort	mergeSort.cu	636	95	100	264	300
transpose	transpose.cu	349	174	100	180	319
scan	scan.cu	290	116	100	90	70
	total	2650	1394	100	1039	1027

[2]The column "hCOV" indicates the statement coverage for the host code. For device code, coverage analysis is a wrong approach as already discussed by the Nvidia community [40].

TABLE III
MUTANT RESULTS FOR EACH MUTATION OPERATOR

category	operator	covered	killed	equiv.	survived	total	MS[3]
C	conditional_boundary_replacement	114	60	45	58	118	0.822
	increment_replacement	12	9	0	3	12	0.75
	logical_replacement	5	2	3	3	5	1
	math_replacement	714	572	86	172	744	0.869
	negate_conditional_replacement	156	130	6	30	160	0.844
	subtotal	1001	773	140	266	1039	0.86
GPU	alloc_decrement	46	38	0	12	50	0.76
	alloc_increment	46	21	0	29	50	0.438
	alloc_swap	46	31	3	19	50	0.633
	atom_removal	1	1	0	0	1	1
	gpu_index_decrement	204	181	0	29	210	0.862
	gpu_index_increment	204	173	0	37	210	0.824
	gpu_index_replacement	403	340	0	71	411	0.827
	share_removal	20	17	0	3	20	0.85
	sync_removal	25	19	1	6	25	0.792
	subtotal	995	821	4	206	1027	0.803

[3]MS represents the mutation score which is calculated by the number of killed mutants divided by the number of non-equivalent mutants (the same in the following tables).

operator based on the number of the generated mutants and the mutation score.

For **RQ2** and **RQ3**, we determine the effectiveness of the mutation operators in assessing test quality of GPU programs based on non-equivalent surviving mutants. Because non-equivalent surviving mutants are crucial to calculate the mutation score, and by investigating non-equivalent surviving mutants, we can see whether those mutants are due to inadequate test suites or not. We have implemented five conventional and nine GPU-specific mutation operators in our tool MUTGPU. We apply all the mutation operators to the six subject systems and manually analyse the non-equivalent surviving mutants. We identify the equivalent mutants by hand.

To compare the conventional mutation operators with GPU-specific ones (**RQ4**), we are interested in what kind of enhancements the GPU-specific mutation operators can bring to the conventional mutation operators. In other words, we would like to investigate whether there exists some bugs or issues that cannot be detected by the conventional mutation operators, but can be detected by GPU-specific mutation operators. Therefore, we first try to manually engineer new test cases to kill all the possible conventional mutants to obtain a C-sufficient test suite for each system. Then, we apply GPU-specific mutation operators to the C-sufficient test suites to see if there are non-equivalent GPU mutants survived. The last step is to manually analyse the remaining GPU mutants that cannot be detected by the C-sufficient test suites.

VII. RESULTS

*1) **RQ1**: frequency of GPU-specific mutation operators & mutation scores:* We sum up the mutant results for each mutation operator in Table III. From Table III, we can observe that operator *gpu_index_replacement* generates the most mutants (411) for GPU programs, followed by operator *gpu_index_decrement* (210) and *gpu_index_increment* (210). This indicates that the GPU indexing operations are commonly used in GPU programming. Thus, designing specific mutation operators for GPU indexing seems necessary. The operator *atom_removal* only produces one mutant. We assume the reason behind the low mutant number (=1) is because the selected subject systems we selected do not contain many atomic operations.

From the aspect of the mutation score, except operator *atom_removal* whose mutation score is 1, the rest ranges from 0.438 to 0.862. This means that not all the mutants generated by the GPU-specific mutation operators can be detected by the existing test suites. There is still space for improvement in the existing test suites. The high mutation score for operator *atom_removal* is due to its low mutant number. This observation indicates that all GPU specific mutation operators we designed are useful in GPU programming to guide the engineers to write better tests.

The GPU-specific mutation operators we propose can be fre-

quently applied in GPU programming. Furthermore, the mutation score obtained from applying these mutation operators ranges from 0.438 to 1.0.

2) RQ2: conventional mutation operators: Table IV summarises the mutant results from running both C and GPU mutants against existing test suites. Overall, there are 1039 C mutants generated in total (as shown in Table IV). 96.3% mutants are covered by the existing test suites (meaning the line from which the mutant is generated is covered by the test suite), while 74.4% mutants (86.0% non-equivalent mutants) are killed. We can see that the mutation score (0.860) is lower than the mutation coverage (0.963), which means some mutants that are covered by the tests can still survive. Looking at the existing test suites, we found that most tests from the six projects do not target the unit-level: the main design of the test suites is to invoke a series of functions in turn and examine the final results in the end. Therefore, it is very likely that a small change in the program (a mutant) do not propagate to the outputs (*fault masking* [41]). Moreover, *test directness*, which measures the extent to which the production code is tested directly, plays an important role in mutation testing [42]. However, there are *few* direct tests in the existing test suite to assess the difference between the original and the mutated programs; this causes a small number of mutants to not be detected by the existing test suites. The comparison of coverage and mutation score indicates that the mutation score is a stronger indicator of test suite quality than test coverage.

Speaking of equivalent mutants, the conventional mutation operators generate 140 equivalent mutants (13.5%) out of 1039 mutants as displayed in Table IV. Furthermore, from Table III where we sum up the mutant results for each mutation operator, we can see that over 50% of mutants (61.4%) are generated by operator *math_replacement*, followed by operator *conditional_boundary_replacement* (32.1%). Most of the equivalent mutants produced by *math_replacement* are because one operand in the math operation is zero, such as *threadIdx.x + 0*, thus the replacement of the math operators, e.g., from $+$ to $-$, does not influence the result. The equivalent mutants from *conditional_boundary_replacement* are owing to the fact that the boundary conditions (the equivalent condition, i.e., $=$) cannot be reached or satisfied. Also, there are a few equivalent mutants guiding us to detect the bugs in the systems. Listing 16 presents an example of a bug caught by the equivalent mutants. The mutant which replaces $<$ to $\leq$ in Line 1 is equivalent to the original program since the variable i does not affect the result of the loop. This equivalency turns out to be a potential bug in the program as variable i is useless in the loop. The cause for the bug is that there is a similar loop in the previous function (shown in Listing 17). Thus, the bug shown in Listing 16 might be due to copy-pasting the previous statements but omitting the modification. This finding supports that GPU programming can benefit from mutation testing.

Moreover, to further understand the effectiveness of the mutation operators in assessing test quality, we need to manually analyse the 126 surviving non-equivalent mutants to see whether these surviving non-equivalent mutants are

```
1  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS){
2    if (xIndex < height && yIndex < width){
3      odata[index]=tile[threadIdx.y][threadIdx.x];}}
```

Listing 16. Bug example of an equivalent mutant (in *transpose.cu*)

```
1  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS){
2    odata[index_out+i] = idata[index_in+i*width];}
```

Listing 17. Cause of bug example in Listing 16

due to inadequate test suites. Compared to conventional CPU programs, testing GPU code is more challenging. Only a small number of mutants (34 out of 126) can simply be killed by improving and adding tests. To detect the remaining mutants, we first need to refactor the existing code, and then add tests. The reason for a number of the mutants that need to be refactored is because many functions cannot be directly accessed from the test suites.

According to the CUDA programming model (shown in Figure 2), there are two parts in the GPU programs: the host (the CPU and the system memory) and the device (the GPU and its memory). The tests are mainly located in the host which invokes a function in the device/host and examines the result. Therefore, if the function is executed on the device and callable from the device only, it is impossible to access the function directly from the host. This exception is the function specified by *__device__*. The workaround is to wrap *__device__* functions with a *__global__* function which can be callable from the host. Other function specifiers, e.g., *static* and *inline*, also prevent the tests to access those functions from a different file. Since the access to *static* and *inline* functions is restricted to the file where they are declared. To test *static* and *inline* functions, we have to remove the *static* and *inline* specifiers to allow the access from a separate test file.

By modifying the function accessibility for the tests and adding direct tests, we can kill another 58 mutants. However, some mutants generated are located in intermediate variables which do not propagate to the output. These are much harder to detect: they usually require to split the method into smaller portions, or refactor the method to non-void. This might considerably alter the structure of the systems, and also affect the mutation score. Therefore, in this study, we leave this type of "stubborn" mutants [43] aside. Except for "stubborn" mutants, to kill a conventional mutant in a GPU program, adding one direct test without carefully choosing test input can work well which we can achieve it within 1 min. Finally, we can achieve 0.962 mutation score by improving the test quality, which indicates the conventional mutation operators can effectively evaluate the existing test quality in the context of GPU programming.

> GPU programming can benefit from the conventional mutation operators which mainly guide the engineers to write direct tests for GPU programs. The average time to engineer a test case to kill a mutant is within 1 min.

3) RQ3: GPU-specific mutation operators: From Table IV, we can see that there are 1027 mutants generated by GPU-specific mutation operators, slightly less than C mutants.

Project	File	C Mutants					GPU Mutants				
		total	covered	killed	equiv.	MS	total	covered	killed	equiv.	MS
MonteCarloMultiGPU	MonteCarlo_kernel.cu	71	65	44	7	0.688	59	59	23	0	0.390
	MonteCarlo_reduction.cuh	14	14	13	1	1.000	2	2	0	1	0.000
convolutionFFT2D	convolutionFFT2D.cu	40	40	40	0	1.000	36	36	27	0	0.750
	convolutionFFT2D.cuh	250	249	213	34	0.986	64	64	61	0	0.953
histogram	histogram64.cu	82	77	70	6	0.921	96	96	81	0	0.844
	Histogram256.cu	48	48	41	5	0.953	81	81	68	0	0.840
mergeSort	mergeSort.cu	264	244	180	33	0.779	300	288	265	0	0.883
transpose	transpose.cu	180	174	99	43	0.723	319	299	231	0	0.724
scan	scan.cu	90	90	73	11	0.924	70	70	65	3	0.970
	total	1039	1001	773	140	0.860	1027	995	821	4	0.803

```
1  if (tid == 0){
2      beta  = 0;beta2 = 0;
3      for (int i = 0; i < blockDim.x; i += VEC) {
4          beta  += sum[i];beta2 += sum2[i];}
5      __TOptionValue t  = {beta, beta2};
6      *d_CallValue = t;}
7  cg::sync(cta);
```

Listing 18. Bug example of an equivalent mutant (*MonteCarlo_reduction.cuh*)

Among all the GPU mutants, 96.9% mutants are covered by the existing test suites. The overall mutation score is 0.803. The same observation holds for GPU mutants as for CPU mutants: the mutation score is higher than the coverage. As we mentioned in Section VII-2, the majority of the tests only examine the final outputs after a series of function calls. Therefore, although a number of mutants are covered by the test suites, their changes do not propagate to the final outputs. This also shows the advantage of the mutation testing over test coverage in evaluating test quality.

In terms of equivalent mutants, GPU specific mutation operators only generate four equivalent mutants. Three are from operator *alloc_swap*, which are all located in the *execution configuration* for a kernel function call (*__global__* function call). The mutants generated by *alloc_swap* are due to the number of threads being the same as the number of blocks in parallel processor allocations. The last equivalent mutant is generated by operator *sync_removal*. This mutant is similar to Listing 16 which indicates the presence of a potential bug (presented in Listing 18). The main task of Listing 18 is to produce a smaller array of the sum result by *reduction*. The code fragment in Listing 18 does not contain write operations to the shared array *sum* and *sum2*, thereby, there is no need for a synchronisation function (*cg::sync(cta)*) in the end to guarantee that all of those writes to the shared arrays complete before anyone tries to read from the buffers. The synchronisation function (*cg::sync(cta)*) in Listing 18 is not necessary. This also confirms that misuses of the synchronisation functions in GPU programming are quite common, and the operator *sync_removal* can well represent common errors in GPU programming. Looking into the surviving GPU mutants, a large number of mutants (131) can easily be detected by modifying the function accessibility (e.g., remove *static* specifer) and adding direct tests as mentioned in Section VII-2. To kill the remaining non-equivalent mutants is more challenging: in addition to examining the expected result after the function call, more factors should be taken into consideration, such as the execution sequence, the size of the test input and the

times of test execution. For instance, to detect the mutants from operator *sync_removal*, it usually requires multiple test executions (> 10 times), as the execution order of the code in GPU cores is undetermined. Although, we assume that GPU cores run the parallel program at the same time, there exist some latencies in different GPU cores. For instance, Thread 1 is executed before Thread 2 in one execution, while Thread 2 is executed before Thread 1 in another execution. Thus, *data races* do not occur every time. Another example is to kill the mutant generated from a *__global__* function. It is very likely to encounter illegal memory access if the test input size is inappropriate. To kill a GPU mutant, it usually requires us to understand the program context very well and choose more than one specific test input to kill a mutant; this could take up to hours to kill a GPU mutant. Thus, those tests designed to kill GPU mutants can better reveal GPU-specific weakness.

However, we found that not all the non-equivalent surviving mutants can be killed by adding test cases. There are 22 mutants not affecting the result of the kernel functions but the GPU performance. The GPU performance means the execution time by GPUs. For example, one mutant generated by *alloc_increment* modifies the number of threads in parallel processor allocations from 256 to (256-1). But the modifications do not influence the function output since the function already takes care of the boundary condition when the test input exceeds the number of parallel threads (just as in Listing 4). Also, the performance difference caused by the allocation decrement (-1) is too small to be sensed by any test. The performance difference is unique to GPU programming since the standard CPU programs do not use GPUs as co-processors. We suggest considering these mutants that only influence the performance at a small scale without output modification as equivalent mutants in the context of GPU programming. Another option to void such "equivalent" mutants generated from GPU programming could be using performance requirement to be part of the definition of a test case passing or failing.

> GPU-specific mutation operators can effectively evaluate the test quality in the context of GPU programming. To kill the GPU mutants, many factors should be taken into consideration, such as test directness, the program context, the execution sequence and the test input size. It takes up to hours to kill a GPU mutant.

```
1  __global__ void padKernel_kernel(float *d_Dst, float *
       d_Src, int fftH, int fftW, int kernelH,int kernelW,
       int kernelY, int kernelX){
2      int y = blockDim.y * blockIdx.y + threadIdx.y;
3      int x = blockDim.x * blockIdx.x + threadIdx.x;
4      if (y < kernelH && x < kernelW){
5          int ky = y - kernelY;
6          if (ky < 0){
7              ky += fftH;}
8          int kx = x - kernelX;
9          if (kx < 0){
10             kx += fftW;}
11         d_Dst[ky*fftW+kx] = LOAD_FLOAT(y*kernelW+x);}}
```

Listing 19. Example of a surviving GPU mutant (*convolutionFFT2D.cuh*)

4) RQ4: conventional vs. GPU-specific: In this section, we are going to shed light on the comparison of the conventional and GPU-specific mutation operators we proposed. As mentioned earlier, we are interested in what kind of improvements the GPU-specific mutation operators can bring to the conventional mutation operators. Thereby, we use C-sufficient test suites (100% mutation coverage for C mutants) as the base to apply mutation testing on the subject systems. Table V displays the mutant results based on the C-sufficient test suites. We can see that C-sufficient test suites have a high average mutation score using conventional mutation operators compared to GPU-specific ones; this is what we expect. Otherwise, GPU mutants are subsumed by C mutants, i.e., if the test suite achieve 100% mutation coverage on C mutants, it also achieves 100% mutation coverage on GPU mutants.

From Table V, we can see that with an increase in mutation score with conventional mutation operators, the mutation score of GPU-specific mutation operators also increases. This is mainly due to the design of the existing test suites, that do not target the unit testing level (mentioned in Section VII-2). We also observe a lack of direct tests for each function in the existing test suites. Thus, when we engineer new tests to kill the C mutants, we mainly concentrate on designing direct tests (as we discussed in Section VII-2). When we target a C mutant, the GPU mutant(s) located in the same line or the same function unit is also under investigation. Once the change(s) of the GPU mutant(s) in the same line or the same function unit can be observed by the direct test designed for the C mutant, the GPU mutant(s) can typically be killed at the same time.

Considering the remaining GPU mutants that are not detected by the C-sufficient test suites, the majority requires more delicate and complex test cases to observe the differences. Take a surviving GPU mutant in File *convolutionFFT2D.cuh* (shown in Listing 19) for example. The function *padKernel_kernel* aims to position the center of the convolution kernel at (0, 0) of the image which makes use of 2D GPU indexing.

All the mutants generated in Line 2 are detailed in Table VI. In order to kill the C mutant (MID = 0) in Line 2, we add a direct test (shown in Listing 20) for function *padKernel_kernel*. However, the newly added direct test cannot detect the two GPU mutants with MID = 2 and MID = 9. Upon investigation, we found in the execution configuration for function *padKernel_kernel*, the numbers of blocks in the two dimensions are the same, i.e., *gridDim.x = gridDim.y =* 1 (see Line 14). Also, the condition in Line 4 of Listing 19

```
1  void test_padKernel_kernel(){
2      int N = 64*64;
3      float *d_Dst,*d_Src;
4      float h_Dst[N],h_Src[N],h_expected[N];
5      for(int i=0;i<N;i++){
6          h_Src[i]=i;
7          h_expected[i]=0;}
8      h_expected[3966]=1.0;
9      ...
10     cudaMalloc((void **)&d_Dst,sizeof(float)*N);
11     cudaMalloc((void **)&d_Src,sizeof(float)*N);
12     cudaMemcpy(d_Src, h_Src, N*sizeof(float),
               cudaMemcpyHostToDevice);
13     cudaMemset(d_Dst,0,N*sizeof(float));
14     dim3 threads( 8,8 );
15   → dim3 threads( 32,8 );
16     dim3 grid(iDivUp(3, threads.x), iDivUp(3, threads.y));
17     padKernel_kernel<<<grid,threads>>>(d_Dst,d_Src,64,64,
               1,1,1,1 );
18   → padKernel_kernel<<<grid,threads>>>(d_Dst,d_Src
               ,64,64, 3,3,3,3 );
19     cudaMemcpy(h_Dst, d_Dst, N*sizeof(float),
               cudaMemcpyDeviceToHost);
20     bool testFlag = true;
21     for(int i=0;i<N;i++){
22         if(h_expected[i]!=h_Dst[i]){
23             testFlag = false;}}
24         ...}
```

Listing 20. Direct test and its improved version for Listing 19

restricts the larger indexes of parallel threads for computation. Therefore, adding the direct tests cannot detect the difference of replacing *blockIdx.y* to *blockIdx.x*. To detect Mutant 2, we need to modify the numbers of blocks in the two dimensions to different values, e.g., set *gridDim.x* = 4 and *gridDim.y* = 1 (see Line 15). Moreover, we need to set the values of *kernelH* and *kernelW* (see Line 17&18) big enough so that the difference of the value *y* can affect the output of *d_Dst* array.

Together with this example, we can see that adding direct tests can kill most C mutants, but not all the GPU mutants. The remaining GPU mutants are more challenging to be killed as their differences with the original program are more subtle to tell. Only given test inputs with specific values and execution settings, these GPU mutants can be detected. This shows that the outputs of the GPU program are easily affected by test inputs and execution configurations. Thus, the corresponding test cases designed according to these types of GPU mutants are of higher quality and can detect more potential GPU-specific bugs in the systems. Therefore, it requires more effort to design delicate tests to kill a GPU mutants than the conventional C mutants.

> Compared to conventional mutation operators, GPU-specific ones are better at guiding the engineers to design more delicate tests to detect the subtle differences in GPU programming. The efforts required to kill GPU mutants are higher than the conventional.

VIII. THREATS TO VALIDITY

External validity: First, our results are based on the CUDA programming model; the results might be different when using other GPU programming models. Second, concerning the subject selection, we chose six GPU projects in total to evaluate our approach. All the projects are benchmark projects from the CUDA SDK [18], and are widely used in the research domain; this can minimise the threats caused

TABLE V
Mutant results (C-sufficient test suites)

Project	File	C Mutants					GPU Mutants				
		total	covered	killed	equiv.	MS	total	covered	killed	equiv.	MS
MonteCarloMultiGPU	MonteCarlo_kernel.cu	71	71	61	7	0.953	59	59	49	0	0.831
	MonteCarlo_reduction.cuh	14	14	13	1	1.000	2	2	0	1	0.000
convolutionFFT2D	convolutionFFT2D.cu	40	40	40	0	1.000	36	36	27	0	0.750
	convolutionFFT2D.cuh	250	249	216	34	1.000	64	64	61	0	0.953
histogram	histogram64.cu	82	77	76	6	1.000	96	96	82	0	0.854
	Histogram256.cu	48	48	42	5	0.977	81	81	69	0	0.852
mergeSort	mergeSort.cu	264	264	213	33	0.922	300	300	280	0	0.933
transpose	transpose.cu	180	180	126	43	0.920	319	319	298	0	0.934
scan	scan.cu	90	90	78	11	0.987	70	70	65	3	0.970
	total	1039	1033	865	140	0.962	1027	1027	931	4	0.910

TABLE VI
Mutant details for Listing 19 in Line 2

MID	operator	details	category	existing	c-sufficient
0	math_rep.	/	c	survived	killed
1	gpu_index_rep.	threadIdx.y	gpu	killed	killed
2	gpu_index_rep.	blockIdx.x	gpu	survived	survived
3	gpu_index_inc.	(blockIdx.y+1)	gpu	killed	killed
4	gpu_index_dec.	(blockIdx.y-1)	gpu	killed	killed
5	math_rep.	-	c	killed	killed
6	gpu_index_rep.	blockIdx.y	gpu	killed	killed
7	gpu_index_rep.	threadIdx.x	gpu	killed	killed
8	gpu_index_inc.	(threadIdx.y+1)	gpu	killed	killed
9	gpu_index_dec.	(threadIdx.y-1)	gpu	survived	survived

by subjects. Moreover, there might exist errors in the GPU hardware and the CUDA toolkit. To avoid this threat, we conduct our experiment on different NVIDIA graphic cards and different releases of CUDA toolkit.

Internal validity: The main threat to internal validity for our study is the implementation of MUTGPU for the experiment. To reduce internal threats to a large extent, we carefully reviewed and tested all code to eliminate potential faults in our implementation. Another threat to internal validity is the detection of equivalent mutants through manual analysis. However, this threat is unavoidable and shared by other studies that attempt to detect equivalent mutants [44], [45].

Construct validity: The main threat to construct validity is the measurement we used to evaluate our methods. We used the percentage of non-equivalent mutants and the mutation score as key metrics in our experiment, both of which have been widely used in other studies on mutation testing.

IX. RELATED WORK

With wide-spread applications of GPUs in High Performance Computing (HPC) [3], [4] and safety-critical domains (e.g., medical science [5] and automotive [6]), there have been increasing attentions on the *quality assurance* of GPU applications [7], [8], such as dynamic analysis [38] and formal verification [37], [39].

Most related to our approach are *fault injection* techniques in GPGPU applications. Farazmand et al. [46] attempted to quantify the Architectural Vulnerability Factor (AVF) of GPU hardware structures using statistical fault injection. They injected faults into register files, local memory, and active mask stack to characterise the vulnerability of different micro-architectural structures in GPUs to soft errors. Yim et al. [47] developed a mutation-based fault injection tool for automated reliability testing of GPU devices. They modeled both single-

and multi-bit errors in the architecture state to represent the silent data corruption (SDC) error.

Fang et al. [48] proposed a fault injection methodology to evaluate the error resilience of the GPGPU applications. They aimed at injecting the faults that represent real *hardware* errors where they adopted the single-bit-flip fault model to simulate transient faults in GPU processors. Hari et al. [49] presented an fault injection-based framework called SASSIFI for GPU application resilience evaluation, especially on soft errors. SASSIFI serves two kinds of tasks: (1) inject bit-flip errors into the register file for AVF analysis; (2) inject errors in the outputs of the instructions for error propagation evaluation.

All the above studies have targeted the errors related to the GPU hardware. While we focus on the software aspect of GPU applications where we design mutation operators in the ways that engineers could make mistakes in GPU programming.

X. CONCLUSION AND FUTURE WORK

This paper aims to explore whether GPU programming can benefit from mutation testing. Compared to the CPU, the GPU differs greatly in the architecture and the programming model, thus, GPU programming comes with its own set of challenges. Upon observation, we found GPU code with issues of memory management, thread management and atomic operations can easily pass the test suite selected with conventional mutation operators. Thus, we propose nine GPU-specific mutation operators according to the main syntactic differences between CPU and GPU programming.

To evaluate our approach, we present a tool coined MUT-GPU and conduct an experiment on six CUDA systems. Our results show promising findings that GPU programming can benefit from mutation testing in three ways: (1) conventional mutation operators can guide engineers to write simple direct tests; (2) GPU-specific mutation operators can lead to more delicate test cases (thus higher quality and more test effort); (3) equivalent mutants can help in bug detection.

Our paper makes the following contributions:

- nine GPU-specific mutation operators;
- a mutation tool (MUTGPU [35]) working on CUDA;
- comparison of conventional and GPU-specific operators;
- a preliminary experiment on six GPU applications [35].

Future work. In the future, we aim to conduct additional case studies on more realistic GPU systems. Also, we would like to explore another GPU platforms, such as OpenCL.

REFERENCES

[1] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2010, version 3.2.

[2] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[3] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 47.

[4] H. Takizawa and H. Kobayashi, "Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing," *The Journal of Supercomputing*, vol. 36, no. 3, pp. 219–234, 2006.

[5] S. S. Stone, J. P. Haldar, S. C. Tsao, B. Sutton, Z.-P. Liang *et al.*, "Accelerating advanced mri reconstructions on GPUs," *Journal of parallel and distributed computing*, vol. 68, no. 10, pp. 1307–1318, 2008.

[6] C. Lee, S.-W. Kim, and C. Yoo, "Vadi: Gpu virtualization for an automotive platform," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 277–290, 2015.

[7] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, "Gpgpus: how to combine high computational power with high reliability," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 341.

[8] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs parallelism management on safety-critical and hpc applications reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 455–466.

[9] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying gpu kernels by test amplification," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 383–394.

[10] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[11] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. on Softw. Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[12] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports quality assurance processes," *Softw. Test., Verif. Reliab.*, vol. 28, no. 6, 2018.

[13] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[14] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[15] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *ICST workshops*. IEEE, 2009, pp. 220–229.

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

[17] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. Int'l Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665.

[18] "CUDA Developer SDK Code Samples," https://www.nvidia.com/object/cuda_get_samples_3.html, [Accessed 03-June-2019].

[19] D. Luebke and G. Humphreys, "How GPUs work," *Computer*, vol. 40, no. 2, pp. 96–100, 2007.

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," 2008.

[21] K. Gregory and A. Miller, *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press, 2012.

[22] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[23] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[24] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[25] "SO29158775: My cuda kernal copying matrix with adjustment is not working," https://stackoverflow.com/questions/29158775/my-cuda-kernal-copying-matrix-with-adjustment-is-not-working, [Accessed 18-March-2019].

[26] "SO25255699: Share memory in CUDA ? How does it CODE work?" https://stackoverflow.com/questions/25255699/share-memory-in-cuda-how-does-it-code-work, [Accessed 18-March-2019].

[27] "SO9488590: Shared memory mutex with CUDA - adding to a list of items," https://stackoverflow.com/questions/9488590/shared-memory-mutex-with-cuda-adding-to-a-list-of-items, [Accessed 18-March-2019].

[28] "SO9859456: cuda thread indexing," https://stackoverflow.com/questions/9859456/cuda-thread-indexing, [Online; accessed 18-March-2019].

[29] "SO21677559: array operation using CUDA kernel," https://stackoverflow.com/questions/21677559/array-operation-using-cuda-kernel, [Accessed 18-March-2019].

[30] "SO33159171: CUDA C sum 1 dimension of 2D array and return," https://stackoverflow.com/questions/33159171/cuda-c-sum-1-dimension-of-2d-array-and-return, [Accessed 18-March-2019].

[31] "SO29233426: Cuda shared memory bug," https://stackoverflow.com/questions/29233426/cuda-shared-memory-bug, [Accessed 18-March-2019].

[32] "SO14057678: cuda matrix multiplication by columns," https://stackoverflow.com/questions/14057678/cuda-matrix-multiplication-by-columns, [Accessed 18-March-2019].

[33] A. Denisov and S. Pankevich, "Mull it over: mutation testing based on llvm," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 25–31.

[34] H. Coles, "PIT Mutation Operators," http://pitest.org/quickstart/mutators/, [Online; accessed 28-May-2019].

[35] Q. Zhu, "Replication Package," https://doi.org/10.5281/zenodo.3484715, [Online; accessed 16-December-2019].

[36] P. McGuire, *Getting started with pyparsing*. O'Reilly, 2007.

[37] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 113–132.

[38] M. Boyer, K. Skadron, and W. Weimer, "Automated dynamic analysis of cuda programs," in *Third Workshop on Software Tools for MultiCore Systems*, 2008, p. 33.

[39] G. Li and G. Gopalakrishnan, "Scalable smt-based verification of gpu kernel functions," in *Proc. ACM SIGSOFT Int'l Symposium on Foundations of software engineering*. ACM, 2010, pp. 187–196.

[40] "Code coverage using NVCC compiler," https://devtalk.nvidia.com/default/topic/980545/code-coverage-using-nvcc-compiler/, [Online; accessed 09-October-2019].

[41] R. Gopinath, C. Jensen, and A. Groce, "The theory of composite faults," in *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 47–57.

[42] Q. Zhu, A. Zaidman, and A. Panichella, "How to kill them all: an exploratory study on the impact of code observability on mutation testing," *PeerJ Preprints*, vol. 7, p. e27794v1, Jun. 2019. [Online]. Available: https://doi.org/10.7287/peerj.preprints.27794v1

[43] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. Intl Conf. on Software Engineering*. ACM, 2014, pp. 919–930.

[44] B. J. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 192–199.

[45] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 74–83.

[46] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based avf analysis of a gpu architecture," *Proceedings of SELSE*, vol. 12, 2012.

[47] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *Int'l Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 287–300.

[48] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 221–230.

[49] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 249–258.