The Journal of Supercomputing
https://doi.org/10.1007/s11227-023-05375-0

Reducing branch divergence to speed up parallel execution of unit testing on GPUs

Taghreed Bagies[1] · Wei Le[2] · Jeremy Sheaffer[2] · Ali Jannesari[2]

Accepted: 3 May 2023

Abstract
Software testing is an essential phase in the software development life cycle. One of the important types of software testing is unit testing and its execution is time-consuming and costly. Using parallelization to speed up the testing execution is beneficial and productive for programmers. To parallelize test execution, researchers can use GPU machines. In GPU applications, multiple threads execute in parallel within a group known as a warp. Branch divergence affects the performance of a warp negatively when some threads run a branch, and the other threads are idle waiting for the first set of threads to finish their execution. In this paper, we propose a novel algorithm to minimize branch divergence when testing an application on a GPU. We arrange test inputs based on the warp size of a GPU machine. Test inputs that have similar control flow paths are grouped within the same warp executing in parallel. Thus, the branch divergence is minimized per warp. We validate and evaluate our algorithm on six benchmarks (57 programs in total). Our approach accelerates the testing execution by up to 3.8x and improves the warp execution efficiency by up to 15x.

Keywords Software testing · Testing in parallel · Branch divergence · CUDA · GPUs

✉ Taghreed Bagies
tbagies@kau.edu.sa

Wei Le
weile@iastate.edu

Jeremy Sheaffer
sheaffer@iastate.edu

Ali Jannesari
jannesar@iastate.edu

1 King Abdulaziz University, Jeddah, Saudi Arabia

2 Iowa State University, Ames, IA, USA

1 Introduction

In the software development life cycle (SDLC), software testing is an important phase and involves 50% of the SDLC, which is time-consuming [1–3]. According to [4], one of the main goals of software testing is to discover program defects before it is put into use. One process of defect testing is unit testing which is the process of testing individual components in isolation. Units may be individual functions. When we apply unit testing, we execute a program using artificial data (test inputs) [4]. However, executing software testing on an application could require seven weeks [5]. Therefore, running software testing in parallel can significantly speed up test execution time [6, 7].

To execute software testing in parallel, some studies use distributed execution environments such as cloud computing or virtual machines (VMs) [6, 8–14]. However, distributed environments are costly with respect to maintenance, energy consumption, and time scheduling for a shared resource [1]. Therefore, two studies use GPUs to parallelize test execution.

When parallelizing the test execution on GPU machines, test inputs will be distributed on multiple threads [1, 15]. Each test input should follow a control flow path of the program under test. Each thread executes a test input, leading to the parallel execution of the program test. However, a control flow path may differ from one test input to another by an instruction in the program under test due to a branching instruction. Hint, each thread may execute different instructions from other threads.

A phenomenon known as branch divergence occurs when threads in a group encounter a branching instruction, not all threads take the same control flow path, which negatively affects the parallelization of test execution on GPU machines. In a GPU, a number of threads execute in parallel within a group known as warp. When parallelizing test execution on a GPU, each test input is executed by a thread with a different control flow path leading to divergent instructions between threads. The execution of divergent instructions will be serialized [1] such that some threads are inactive, waiting for other threads to finish their execution. As a result, the branch divergence among threads in a warp increases the overall test execution time.

To address branch divergence in GPU general applications, some researchers have proposed methods that change the source code of an application [16–28]. These proposed methods are not applicable for software testing since we must not change the source code for a program under test.

In this paper, we propose a novel algorithm to minimize the branch divergence among threads per warp to test a program in parallel using a GPU machine. Essentially, we collect the branch traces of the program from its test inputs. Then, we generate a similarity matrix (a complete graph) for each pair of test inputs by utilizing Euclidean distance. After that, we build the minimum spanning tree (MST) of the complete graph. If the number of connected nodes in the generated MST is greater than or equal to the size of a warp in a GPU, we sort the connected nodes based on the weights of their edges. We use the warp size to determine the number of sorted nodes to store in a bucket and remove from the complete graph. We repeat this step until the graph is empty. The result of this step is a list of buckets, each with a fixed size (warp size) of test inputs. From these buckets, we produce a set called arranged test inputs (ATI) that can be used to test a program with minimal branch divergences.

We evaluate the algorithm on six different benchmarks, including 57 programs in total. The results show that our algorithm speeds up the testing execution of four out of six benchmarks up to 3.8x. It also improves the warp execution efficiency up to 31.98 threads/warp and warp non-predicated execution efficiency up to 99.98%.

In summary, this paper makes the following contributions:

- A novel algorithm to arrange the test case inputs reducing the branch divergence within a warp on GPUs,
- An algorithm to generate traces from source code that could be used for multiple software testing purposes,
- Empirical evaluations on different domains of benchmarks for correctness and performance improvement, and their results.

This paper is organized as follows: Sect. 2 provides background information related to GPU machines. Section 3 explains a motivation example of how branch divergence emerges in test execution on GPUs. Section 4 presents the literature review. Section 5 describes the algorithm of reducing branch divergence. In Sect. 6, we describe how we evaluate the algorithm. Section 7 analyzes the results. Section 8 is the discussion. Section 9 highlights threats to validity. Section 10 concludes and highlights future work.

2 Background information

To parallelize the test execution on GPU machines, testers need to write a test driver by using a GPU programming model and distribute test inputs on multiple threads [1, 15]. These threads will be executed within a warp. To understand the concept of threads and blocks on GPUs as well as GPU warp, this section provides essential information regarding GPUs and their programming models.

2.1 GPU: programming models

A tester can use CUDA programming models to implement a test suite on GPUs. CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel computing engine on GPUs to solve many complex computational problems more efficiently than on a CPU. CUDA allows a developer to define C++ functions known as kernels. When a kernel is called, it runs N times in parallel by N different CUDA threads instead of only once, like regular C++ functions [29].

2.2 GPU: threads and blocks

According to Gupta [30], CUDA provides an abstraction of GPU architecture acting as a bridge between an application and its possible implementation on GPU hardware. In this abstraction, a group of threads is called a CUDA block. CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of threads.

Since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core, the number of threads per block is limited. On current GPUs, a thread block may contain up to 1024 threads. Nonetheless, a kernel can be executed by multiple equally shaped thread blocks. The total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. Line#4 specifies the number of threads per block and the number of blocks per grid [29].

```
1 int main(){
2     int threads = 32;
3     int blocks = 1;
4     kernel<<<blocks, threads>>>(A,B,C);
5 }
6 __global__ void kernel(float* A, float* B, float* C){
7     int i = threadIdx.x;
8     C[i] = A[i] + B[i];
9 }
```

Listing 1: A sample CUDA code that uses the built-in variable threadIdx, adds two vectors A and B of size N, and stores the result into vector C

Listing 1 shows a CUDA sample code that adds two vectors A and B of size N and stores the result into vector C. A kernel is defined using the __global__ declaration specifier (line#6). The number of CUDA threads that execute that kernel for a given kernel call is specified using <<< ... >>> a new execution configuration syntax (line#4). Each thread that executes the kernel is provided by a unique thread ID accessible within the kernel through built-in variables (threadIdx in line#7). The threadIdx variable is a 3-component vector. Thus, threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume [29].

2.3 GPU warp

A GPU architecture is built around a scalable array of multithreaded streaming multiprocessors (SMs). A multiprocessor is designed to execute hundreds of threads concurrently. GPUs employ a unique architecture called SIMT (single-instruction, multiple-thread) to manage a large number of threads. In SIMT, a multiprocessor creates, manages, schedules, and executes threads in groups of 32

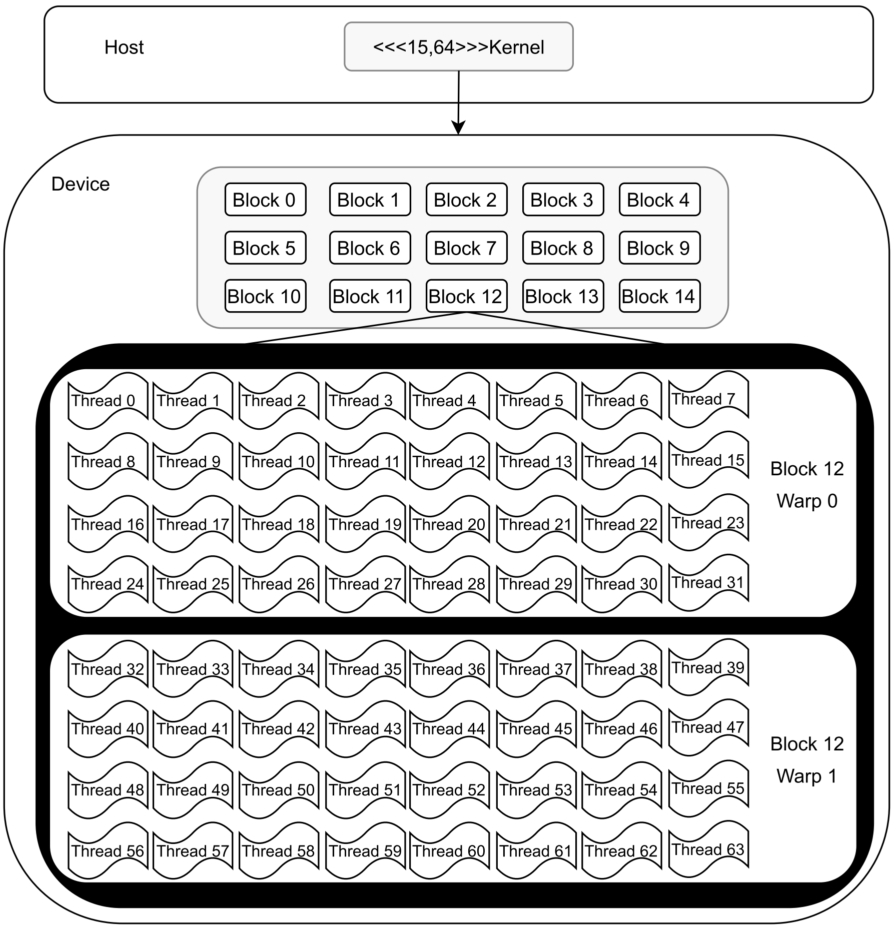


Fig. 1 An example of GPU blocks,threads, and warps

parallel threads called warps. Individual threads start together at the same program address in a single warp, but they have their instruction address counter and register state. Therefore, a group of threads is free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology [29].

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps. Each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs, with the first warp having thread 0 [29].

As shown in Fig. 1, a programmer specifies 15 blocks and 64 threads per block when calling a kernel. A block is divided into warps (groups of 32 threads). A warp is the scheduled unit. The threads of the same block are executed in a given core warp by warp in a SIMD (single-instruction, multiple-data) fashion.

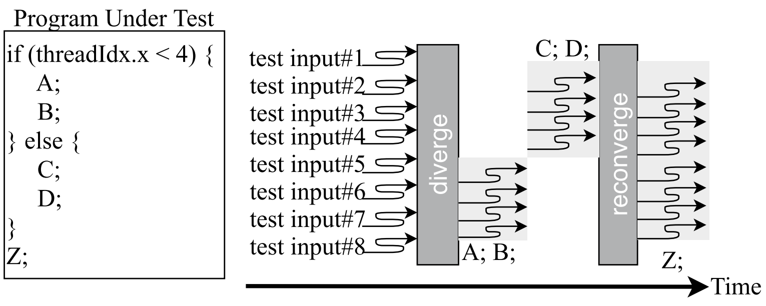


Fig. 2 An example of a control flow divergence problem when parallelizing unit testing on a GPU machine

A warp executes one instruction at a time. Thus, full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths [29].

3 Motivation example

A GPU has been used as an accelerator for different applications [31] and applied in software testing [1]. It achieves high performance by taking advantage of a warp executing a group of threads in single-instruction, multiple-thread fashion [32]. The group of threads within a warp must execute the same instruction at the same time. In other words, threads cannot diverge within a warp because branch divergence will result in serial execution, which can result in a significant performance loss [33].

Figure 2 shows an example to test a program on a GPU machine. There are eight test inputs distributed in eight threads within a warp. When executing the if–else statement, four threads execute the if part (A and B) whereas the other four threads execute the else part (C and D). The warp executes the if part and then proceeds to the else part. While executing the if part, all threads (of test#1-test#4) are inactivated. When execution proceeds to the else part, all threads (of test#5-test#8) are inactivated. Therefore, the if and else parts are executed sequentially not in parallel.

Our main purpose is to reduce the number of inactive threads per warp by arranging test inputs such that test inputs that have similar control flow paths should be grouped in the same warp. Test inputs that have dissimilar control flow paths should be executed in different warps. As a result, the number of inactive threads per warp will be reduced and eventually improve the performance of testing.

If we arrange the test inputs to reduce branch divergence, these arranged test inputs could be used several times during the SDLC. The test execution is repeated several times when a change is made to a program to fix a bug [34, 35]. Changing source code does not usually require changing its corresponding test because there is a weak correlation between the two [36]. According to Rothermel et al. [37], developers often save the tests they develop for their program to reuse later as the program evolves.

4 Related work

The previous literature related to parallel software testing falls into four categories: (1) parallel test execution on a GPU by implementing a test suite using a GPU programming model and reading test case inputs from a file, (2) parallel test suite execution (e.g., using an existing framework such as JUnit) in cloud computing or VMs, (3) parallel test suite execution in a multicore CPU, and (4) parallel test generation. For branch divergence in GPU applications, the previous literature falls into two categories: (1) hardware-based techniques, and (2) software-based techniques.

Our study is related to the first category. The most related work implements a test suite that reads test inputs from a file and uses GPU machines to run the testing in parallel [1, 15]. In [15], they use CUDA to implement a test driver and run it in parallel for embedded software. Yaneva et al. [1] propose a compiler-assisted framework to automatically generate an OpenCL code from a C sequential program and execute the tests in parallel on a GPU machine. Both studies use EEMBC (an automotive benchmark) for evaluation and highlight the control flow divergence issue as a topic for future work. In this paper, we address this problem and propose an algorithm to arrange test inputs reducing the branch divergence per warp such that it decreases the number of inactive threads per warp. It improves the warp execution efficiency that should speed up the process of unit testing on GPUs. To evaluate our algorithm, we use EEMBC benchmark used in [1, 15] in addition to benchmarks from other domains.

With regard to the second category, some researchers use cloud computing or VMs to execute a test suite in parallel [6, 8–14]. However, a test suite may contain an order-dependent test [38]. This could affect the results when parallelizing the execution of a test suite [39]. Therefore, some researchers have proposed algorithms and studied the impact of these dependencies when applying parallelization in a test suite on cloud or VMs [7, 39, 40]. On the contrary, we use GPU machines and an existing test inputs set. Each test input represents a possible value that can be used to test a program. Each test input will have a different control flow path coverage from the other test inputs. When we parallelize the test execution, we distribute test inputs on different threads. There will be no shared data between different threads, so there is no dependency between different test inputs.

The third category is parallel test execution on a multicore CPU [41]. They presented a parallel implementation of the adaptive testing (AT) technique that improves the efficiency of traditional random/partition testing. Unlike their approach, our study compares the use of a single computer of multiple threads with a GPU for parallel test execution. We focus on addressing the problem of branch divergence to decrease test execution time on a GPU.

The last category in parallel software testing is related to parallel test generation [42, 43]. The studies in this category are different from our work in that they

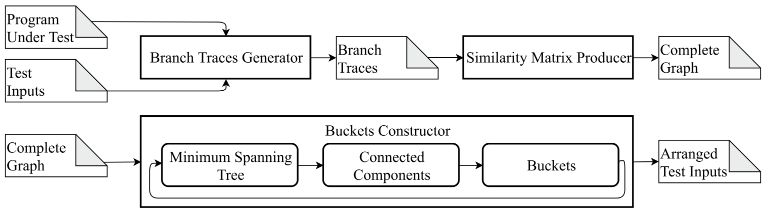


Fig. 3 Steps of the overview of the approach

produce algorithms that use Korat (an existing tool for test generator) and run in parallel to generate test inputs, whereas we use existing test inputs (generated by developers/tools) and arrange them to parallelize the test execution in a GPU machine.

With regard to branch divergence in GPUs' applications, some researchers propose hardware-based techniques [44–53]. Although these techniques are able to minimize the branch divergence for general-purpose GPU applications, they require hardware support. However, our study addresses branch divergence for parallel test execution without requiring changing GPU hardware configurations.

Other researchers introduce software-based techniques to handle branch divergence in general-purpose GPU applications [16–28]. However, these techniques require changing source code to make it run in parallel. In software testing, it is required to not change the source code of an application. Therefore, these techniques could not be applied to handle branch divergence in parallel test execution on a GPU.

To the best of our knowledge, this is the first work on minimizing branch divergence to speed up the execution of testing on GPU machines.

5 Approach

When testing a program on GPUs, a single warp should execute similar control flow paths of test inputs to reduce the branch divergence. In other words, a warp should run a cluster of test inputs that have similar control flow paths. Each cluster must contain 32 test inputs (the warp size). Although we can consider branch divergence concerning test execution on GPUs as a clustering problem, most of the clustering algorithms do not specify the number of elements in each cluster. Therefore, we propose a new algorithm that groups different test inputs based on their control flow paths into a number of clusters. Each group has 32 test inputs.

Figure 3 shows an overview of our approach. First, we use existing test inputs to generate branch traces of a tested program. Since the branch divergence happens at the source code level, our clustering algorithm uses the source code to trace the control flow path of each test input. Branches (if statements and number of loop iterations) are the basic unit that differs a control flow path of a single test input from the others. To trace, we use a vector of branches in a source code and count how many a branch occurs for each test input.

```
Algorithm 1: Arranging Test Inputs
Input: program under test(p), test inputs(t), number of tests(n)
Output: Arranged Test Inputs (ATI)
Function TracesGen($p$, $t$, $n$):
    $max \leftarrow 0$
    $no\_branches \leftarrow getNo\_Branches(p)$
    traces[n][no_branches]
    for $testID = 0 \rightarrow n$ do
        $counter \leftarrow 0$
        $line \leftarrow readLine(p(t[testID]))$
        while line is not Empty do
            execute(line)
            if line is a branch then
                $branchID \leftarrow getBranchID(line)$
                if line has a function call then
                    $traces[testID][branchID] \leftarrow -1$
                    counter++
                else
                    $traces[testID][branchID] \leftarrow$ (++counter)
            $line \leftarrow readLine(p(t[testID]))$
        if max¡counter then
            $max \leftarrow counter$
    for $tID = 0 \rightarrow n$ do
        for $bID = 0 \rightarrow no\_branches$ do
            if $traces[tID][bID] == -1$ then
                $traces[tID][bID] \leftarrow max * no\_branches$
    return traces
Function SMGen($traces$,$n$):
    sm[n][n] //Similarity Matrix
    for $i = 0 \rightarrow n$ do
        for $j = i \rightarrow n$ do
            $sm[i][j] \leftarrow EuclideanDist(traces[i], traces[j])$
            $sm[j][i] \leftarrow 0$
    return sm
Function BucketsGen($sm$,$bl$,$t$,$n$): //buckets' list(bl)
    $graph \leftarrow sm$
    $warpSize \leftarrow 32$
    if graph is empty then
        $no\_buckets \leftarrow n/warpSize$
        $ArrangedTestInputs \leftarrow set()$
        for $i = 0 \rightarrow no\_buckets$ do
            for $j = 0 \rightarrow warpSize$ do
                ArrangedTestInputs.add(t[bl[i][j]])
        return ArrangedTestInputs
    $MST \leftarrow list()$
    for (u, v) in graph.edges ordered by weight(u, v) do
        if MST.connectedComponents >= warpSize then
            sortByWeight(MST)
            addFirstNodes(bl,MST,warpSize)
            removeFirstNodes(MST,graph,warpSize)
            BucketsGen(graph,bl,t,n)
        else
            addEdge(MST,(u,v))
```

Then, we compute the Euclidean distance of each pair of branch traces, which produces the similarity matrix. A clustering algorithm usually uses a matrix to find similarities between points, such as cosine similarity and Euclidean distance. Euclidean distance accounts for magnitude while cosine distance does not [54, 55]. Since the magnitude of the vectors (branches in the source code) is critical, our algorithm uses the Euclidean distance measurement. As a result, the algorithm builds the traces of each test input based on their source code branches and calculates the similarity matrix using the Euclidean distance between each pair of test inputs.

We consider the similarity matrix a complete undirected graph to group the similar test inputs in a cluster. In the graph, each node represents an ID of a test input. Since an MST of an undirected graph is a connected subgraph covering all the graph nodes with the minimum possible number of edges (distances), the last step of our algorithm is to divide the complete graph into several MSTs.

At each iteration of creating an MST, if the number of connected nodes in the MST is equal to or greater than the warp size (32 threads/warp in GPUs [56]), we sort these connected nodes. Then, we remove the first 32 connected nodes from the graph and store them in a bucket. Each bucket has a fixed number of test inputs (32). We repeat generating an MST, finding the 32 connected components, removing them from the graph, and storing them in a new bucket until the graph is empty. Note that the smaller the distance between nodes, the more similar their control flow paths are.

As a result, there will be several buckets, each with a fixed number of test inputs. The number of buckets equals the number of test inputs divided by the number of test inputs in each bucket. A warp will execute each bucket in a GPU machine. In the end, a group of 32 test inputs will be executed by a group of 32 threads/warp. We summarize our approach in Algorithm 1. We will discuss each step in detail in the following sections.

5.1 Branch traces generator

For each test input of a program under test, we collect traces of its control flow path with respect to branches. The two important factors to distinguish one test input from the others in terms of branches are: (1) which branches are executed by a test input, and (2) in which order those branches are executed (e.g., branch#4 executes before branch#5 for test#1, branch#3 executes before branch#5 for test#2).

To represent which branches are executed for each test input, we use 2D arrays (line#4) such that the row represents the number of test inputs and the column represents the total number of branches in a program under test. The number of branches is collected statically from a program under test, so it is a fixed number for all test inputs (line#3).

To keep track of the order of branches, we use a counter variable. For each test input, we initialize the counter with zero (line#6). Each time a branch is executed, we increment the counter and assign its value to the array's element of the branch (line#16). Note that the counter variable will have different values for different test inputs.

In a GPU machine, each warp should execute the same function under test with different test case inputs. Since having different tested functions per warp will increase its branch divergence, we should add a bigger weight for a branch having a function call than other branches. The value of the counter helps to indicate the biggest value a branch has. We keep track of the maximum value of a counter from different test inputs by using the max variable (line#2,19). Since the value of max

Table 1 An example of the output of branch traces generator step for three test inputs. The number of test inputs is 1024. The number of branches in a program under test is four. The value represents how many times a branch executes

Input#	Branch#1	Branch#2	Branch#3	Branch#4
1	3e6	4e6	4e6	0
2	0	291	3e4	3e4
3	0	0	98	3e4

is unknown until the last test input executes, we assign a negative one to the visited element of a branch having a function call (line#13). After executing all test inputs, the value of this branch will be the total number of branches multiplied by the maximum value of the counter (line#22, 23). As a result, the weight of this branch will be the biggest value among other branches.

The input of this step is the source code of a program under test, test inputs, and the number of test inputs. The output is the 2D array (traces) of the size number of test inputs times the number of branches. Each row contains numbers representing how many times a branch has been visited by a test input. Table 1 shows an example of the output from this step.

5.2 Similarity matrix producer

From the previous step, we have a 2D array indicating how many times a branch has executed for each test input. We use this array to build the similarity matrix. The distance between each pair of test inputs is important. For example, the test input#1 takes branch A and branch B. The test input#2 takes branch B and branch C. The test input#3 takes branch C and branch D. The test input#1 is similar to test input#2 by taking branch B, and test input#2 is similar to test input#3 by taking branch C. However, the test input#1 and #3 are dissimilar. Therefore, the similarity matrix will be beneficial to take care of this situation.

We utilize the Euclidean distance to compute the distance between each pair of test inputs (line#29). For instance, we compute the distance between (input#1, input#2), (input#1, input#3), (input#2, input#3) and so on. Since the distance of (input#1, input#2) equals to the distance of (input#2, input#1), we do not calculate the distance twice (line#30). The below equation shows an example of the Euclidean distance for input#1 and input#2 from Table 1.

$$\text{EuclideanDist}(\text{input }\#1,\ \text{input }\#2) =$$
$$\sqrt{(3e6-0)^2+(4e6-291)^2+(4e6-3e4)^2+(0-3e4)^2} \approx 6e6$$

Table 2 shows an example of the output result (the similarity matrix) of the values shown in Table 1. The input of this step is the 2D array (Traces) generated from the Branch Traces Generator step and the number of test inputs. The output of this step is a matrix with the size of $(\text{number\_of\_test\_inputs})^2$.

Table 2 Similarity matrix representing the distance between a pair of test inputs for Table 1

Test input	Input#1	Input#2	Input#3
Input#1	0	6384314.535	6403133.296
Input#2	0	0	29903.41594
Input#3	0	0	0

5.3 Buckets constructor

The previous step generates the similarity matrix representing a complete undirected graph. Each node represents a test input, and the edges' weight represents the distance between nodes. The distance represents how similar one test input is to the others in terms of its control flow paths.

5.3.1 Minimum spanning tree (MST)

Utilizing the MST is beneficial to select similar test inputs. We use Kruskal's algorithm since it builds the MST by sorting the edges' weights (distances) and adds nodes based on the smallest weight of edges (line#43,50). This guarantees that the generated MST at any given time has edges with the lowest weight. Note that the smallest edge's weight provides the most similar pair of test inputs.

As shown in Table 2, the edge's weight between Input#1 and Input#2 is 6384314.5 whereas the edge's weight between Input#2 and Input#3 is 29903.4. Since the edges' weights are ordered, the algorithm will add first the edge between Input#2 and Input#3 to the MST. Note, Table 1 shows Input#2 is more similar to Input#3 than to Input#1 (e.g., not executing Branch#1 and executing Branch#4 for 3e4 times).

5.3.2 Connected components

Since Kruskal algorithm builds the MST by adding the smallest edge's weight in each iteration, there might be several unconnected components. At the first iteration, for example, it might add the edge between node#1 and node#3. In the second iteration, it might add the edge between node#2 and node#4. There might not be an edge between node#1 and node#2; or an edge between node#3 and node#2. There might not be an edge between node#1 and node#4; or an edge between node#3 and node#4. Thus, there are two unconnected components such that node#1 and node#3 are similar but they are different from node#2 and node#4 at this iteration of building the MST.

Therefore, at every iteration of adding a new edge to the MST, we examine if it has connected components equal to the warp size because these connected components represent test inputs that are similar in their control flow path. They should be executed in the same warp. If the MST has a number of connected components more than the warp size, we sort the connected components by their edges' weights and use only the first warp size connected components (line#44,45).

5.3.3 Buckets

When finding and sorting the connected components in the MST, we store the first warp size nodes in a bucket (line#46). Then, we remove them from the graph (line#47). We repeat the steps of building MST, checking the number of connected components, putting nodes into buckets, and removing nodes from the complete graph until the graph is empty (line#48,35). Note that when the number of connected components is bigger than the warp size, the excess nodes will not be removed from the graph; instead; they will be used in the second iteration of the algorithm.

In the end, we have a list of buckets such that the number of buckets is the number of test inputs divided by the warp size (line#36). Each bucket contains IDs of test inputs that are grouped based on their similarity to the branch traces. An ID of a test input helps to find the actual data of each test input. Thus, we use these buckets to create ATI (line#40).

6 Evaluation

To reduce the number of inactive threads per warp when testing a program on a GPU, we consider the distance between different test inputs based on their control flow divergences. Test inputs that have similar control flow paths will have a small distance in terms of branches. The purpose of our algorithm is to arrange test inputs by grouping similar test inputs based on their branch distance.

The evaluation of our algorithm is to answer the following Research Questions (RQs):

1. RQ1: Is our algorithm valid and correct in clustering similar test inputs based on their control flow paths?
2. RQ2: Is our algorithm able to speed up the test execution on a GPU machine? (comparing the execution time of ATI and Random Test Inputs (RTI) with respect to a sequential version.)
3. RQ3: Is our algorithm able to reduce the branch divergence? (measuring the effectiveness of the approach by using (1) warp execution efficiency, (2) warp execution non-predicated efficiency, and (3) stall memory dependency?

6.1 Experimental design

The experiment aims to evaluate our algorithm that arranges test inputs of a program based on their execution path (branch traces). We examine if our approach correctly arranges a set of test inputs, decreases the execution time on GPUs compared to CPUs, and reduces branch divergence. All the implementation and results are available on our GitHub repository text.[1]

The first research question (RQ1) is to validate that our algorithm arranges test inputs as expected. In other words, does our algorithm sort test inputs as excepted? Does our algorithm lose any test inputs while sorting the test inputs? Does our algorithm duplicate any test inputs?

To answer RQ1, we need three sets: (1) inputs set, (2) expected results set, and (3) actual results set. Each set has the same 1024 test inputs (see Sect. 6.3) and is divided into 32 groups. Each group contains 32 test inputs. The first set has unarranged test inputs. The second set has well-arranged test inputs. The third set is created by applying our algorithm to the first set and compared with the second set.

Since the arrangement is based on the control flow paths of test inputs, we cannot know the well-arranged test inputs in advance. To guarantee the arrangement, we will duplicate test inputs. We will construct the 1024 test inputs as follows: (1) generate 32 test inputs (see Sect. 6.3 for how we generate test inputs) and (2) repeat each test input 32 times. The first set (inputs set) has each group with different test inputs, which guarantees that this set is not well arranged. In the second set (expected results set), every group will have the same test inputs and consider well-arranged test inputs. Then, we use our algorithm to arrange the first set and build the third set (actual results set). We can conclude that our algorithm is capable of arranging the set as expected by comparing the actual results set with the expected results set.

To answer the second and third research questions, we select six different benchmarks and generate two different sets of test inputs (Random and Arranged) for each benchmark. RTI is generated randomly while ATI is generated by using our algorithm. For each benchmark, we implement a test driver to test a benchmark on a CPU and GPU machine. For a GPU machine, we run the test driver with ATI and RTI.

For CPU, we use Intel®Xeon®Gold 6140 Processor and implement a test suite in C. We use clock_gettime() [57] to measure the execution time in seconds. For a GPU machine, we use NVIDIA Volta architecture Tesla V100. We implement the test driver by using CUDA that is capable of programming multithreaded GPUs and scales transparently to hundreds of cores [58]. We use cudaEventCreate() [59] to calculate the execution time in seconds.

6.2 Benchmark

A GPU is applicable to handle data-parallel computations such that it executes the same program on many data elements in parallel [60]. In data-parallel processing, we map data elements to parallel processing threads. To speed up the computations,

[1] https://github.com/tbagies/GPU-BranchDivergence.

many applications processing large data sets can use a data-parallel programming model such as CUDA in GPUs. Running program testing in parallel matches the concept of data-parallel processing. To test a program, we need to execute it with different test inputs several times. Therefore, GPUs are suitable to run testing in parallel such that each test input will be executed by a thread running concurrently.

Although some existing tools create a test suite for a specific programming language, currently no tool can automatically generate a test suite in a data-parallel programming model for a program implemented in C to be executed on a GPU. Therefore, we design our experiment similar to the state of the art [15] and use CUDA that is a widely used programming model in GPUs to implement a test suite for a list of benchmarks written in C.

CUDA does not support some C standard libraries [60] and recursive function. We cannot execute a program testing that deals with string or reads from a file and writes to a file. We must not modify the source code of a program under test; we must test it as it is provided. Unfortunately, these limitations do not allow us to use large-scale applications. All the C real-world applications rely heavily on C standard libraries (String, read from a file, Math library, etc.). Thus, we could not test our algorithm with large-scale applications.

We consider these limitations and create a set of criteria to choose benchmarks. The first criterion is that the source code of a tested program does not rely on the C standard libraries such as string comparison. The second criterion is that inputs of a program should be provided by its developer or easy to generate automatically. The last criterion is that selected programs are different in their functions, input types, number of inputs, and source code structure such as control flow. Table 3 summarizes the list of selected benchmarks based on our criteria. Each benchmark includes several programs that we consider as functions under test (FUT).

Polybench is a benchmark suite of numerical computations with static control flow in various application domains such as linear algebra computations and data mining. Its source code is available in [61] and it involves nested loops and handles arrays with a different number of dimensions and computations. We choose two programs from different domains based on our criteria. In addition, we use EEMBC and select the seven functions used by [1, 15].

We use GitHub which is a popular repository and an easy way to obtain source code [62]. We choose Image Manipulation Application (IMA) since it has many functions, many if statements, and a combination of "while" and "for" loops [63]. Also, we choose C-Sorting Library (SortLib) [64] to test our approach on an existing library and evaluate the benefit of our approach to create the unit testing for libraries. Additionally, we use two different benchmarks related to well-known algorithms: graph algorithm (GAlg) [65] and dynamic programming (DynProg) algorithms [66]. We test only the functions that satisfy our criteria from these benchmarks.

6.3 Test inputs

We use the warp size (32 in Tesla V100) to specify the number of threads (i.e., we can have the minimum of $2^n (n >= 5)$. Since, in this GPU machine, the maximum

Table 3 List of selected benchmarks

Image manipulation application (IMA)	C-Sorting (SortLib)	Polybench	Dynamic programming algorithms (DynProg)	Graph algorithms (GAlgo)	EEMBC
Sobel_operator	Bubble	Adi	Binomial Coeff	BFS	A2time01
Prewitt_operator _3by3kernel	Quick	Atax	Matrix Chain Order	Dijkstra	Aifftr01
Prewitt_operator _5by5kernel	Merge	Correlation	Fibonacci Numbers	Floyd Warshall	Aiifftt01
Vertical_operator	Odd Even	Covariance	Kadane's Algorithm	MST	Idctrn01
Horizontal_operator	Cocktail	Durbin	KnapSack		Puwmod01
Diagonal_compass_nw_operator	Comb	Folyd_marsharl	Is-Subset-Sum		Rspeed01
Diagonal_compass_ne_operator	Gnome	Jacobi-2d-imper	Longest Common Subsequence		Tblook
Roberts_operator	Insertion	Ludcmp	Edit Distance		
Gradient_filter	Shell	Reg_detection	Egg Dropping Puzzle		
Increase_saturation	Selection	Syrt2k	Optimal Strategy Of Game		
Increase_intensity	Radix		Longest Increasing Subsequence		
	Pancake				
	Heap				

Table 4 Generating random test input set for each benchmark

Benchmark	Steps
IMA	(1) Selects images with different size from a local directory
	(2) Specifies FUT
	(3) Generates random numbers (inputs of FUT),e.g., the amount of rotation
SortLib	(1) Generates random numbers of different array sizes
	(2) Generates random numbers to be stored in the array
	(3) Specifies FUT
Polybench	(1) Generates random numbers of different array sizes
	(2) Uses functions provided by the developer of the Polybench to generate floating point numbers and stored them in the array
	(3) Specifies FUT
DynProg	(1) Generates random numbers of different array sizes
	(2) Specifies FUT,
	(3) Generates random data to be stored in an array data might be a letter or number depends on FUT
GAlg	(1) Generates different types of graphs with different number of nodes and edges by using Python networkx: Complete, Dense, undirected, and directed graph
	(2) Specifies FUT
EEMBC	(1) Generates the input from the code provided by the developer of EEMBC
	(2) Specifies FUT

number of threads per block is 1024 [56], the minimum number of test inputs could be 1024 test inputs. Therefore, we use 1024 test inputs for each benchmark and distribute the 1024 test inputs in several blocks based on the warp size (32 threads/warp). As a result, the number of threads per block is 32, while the number of blocks is the number of test inputs (1024) divided by the number of threads per warp (32).

To generate a 1024 RTI, we use test inputs provided by a developer of each selected benchmark. If a developer does not provide test inputs, we look at the specification of each program to generate the inputs. Table 4 shows how we generate the test inputs for each selected benchmark. The output of this step is six sets of RTI (i.e., each benchmark has its RTI).

To produce ATI for each selected benchmark, we apply our algorithm to each RTI. The algorithm groups the 32 similar control flow paths of test inputs within a bucket (block on a GPU machine). There are 1024 test inputs. Our approach will produce 32 buckets executed within 32 blocks in parallel. Each has 32 test inputs executed within 32 threads per warp in parallel. The output of this step is ATI for each selected benchmark.

6.4 Test driver

For each selected benchmark, we implement two versions of a test driver. The first version is the sequential version to run the test on a CPU machine by using C language. The second version is to execute the test on a GPU machine in parallel by using CUDA.

The test driver has three parts: (1) a data structure to store test inputs and output results, (2) a function reader to read test inputs from a file, and (3) a function to launch the unit testing. The first and second parts are the same in the two versions (CPU and GPU).

```
 1 int n=1024 //Number of test inputs;
 2 typedef struct{
 3     int length; //size of the enetred array
 4     char typeOfSort; // indicating FUT
 5 }inputData; // input structure for program under test
 6
 7 typedef struct{
 8     int *pArrayToSort;
 9 }outputData;  // output results after testing the program
10
11 void readingInput(inputData *dataIn, outputData* dataOut){
12     // read line from test inputs file
13     // to store the data in dataIn
14     // use malloc to allocate dataOut.pArrayToSort and store its elements}
15
16 //FUT
17  void bubbleSort(int * pArray, int howMany);
18  void quickSort(int * pArray, int start, int end);
19
20 main(){
21     inputData *dataIn = (inputData*)malloc(sizeof(inputData)* n);
22     outputData *dataOut = (outputData*)malloc(sizeof(outputData)* n);
        readingInput(dataIn,dataOut);
23     TestDriverSequential(dataIn,dataOut) // Run on CPU
24     TestDriverParallel(dataIn,dataOut) // Run on GPU }
```

Listing 2: An example of pseudocode of input and output structure and a function reader.

6.4.1 Data structure

To simplify reading the test case inputs, we use Array of Struct (AoS) to store each input such that each member in the struct represents a data of a test input (similar to [1]). If, for example, a program receives an integer number as an input, we create a struct that has these input types (Listing 2 line#2–4). Also, we add the char variable (action) to indicate which FUT will be executed (Listing 2 line#5). In the main method, we create an array of size equal to the number of test inputs of this struct (Listing 2 line#21).

Similarly, we define and declare AoS for the output values (Listing 2 line#8–10, and line#22 respectively). We have two AoS for input and output similar to [1]. This way reduces the amount of data that will be transferred from the device to the host. For example, we will only transfer the array result from the device to the host memory instead of transferring all values of the variables from input and output struct.

6.4.2 Test inputs reader

To read the inputs' values, we implement a function called readingInput (Listing 2 line#12). It receives the inputData and outputData as arguments. Then, it reads an input file that has test inputs' values. Next, it stores the values of each test input in the inputData as well as outputData.

```
 1 TestDriverSequential(InputData *dataIn, OutputData *dataOut){
 2   for(i=0; i<n; i++)
 3     switch(dataIn[i].typeOfSort){
 4         case 'b' :
 5           printf("Using bubblesort\n");
 6           bubbleSort(dataOut[i].pArrayToSort, dataIn[i].length);
 7           break;
 8         case 'q' :
 9           printf("Using quicksort\n");
10           quickSort(dataOut[i].pArrayToSort, 0, dataIn[i].length - 1);
11           break; } }
```

Listing 3: Part of the SortLib test driver pseudocode for sequential version.

6.4.3 Launch test inputs

This is the main different part between the sequential (on a CPU machine) and the parallel (on a GPU machine) versions. For the sequential version, we add a loop iterating on the number of test case inputs. Inside the loop, we invoke all functions under test. Listing 3 (line#2–9) shows an example of a test driver for functionA() and functionB().

For the parallel (GPU) version, we specify the number of threads per block, as well as the number of blocks per grid (Listing 4 line#2–3). As discussed above, the number of threads is 32 (warp size) per block. Then, we allocate data in the device memory and transfer the data from the host memory to the device memory (Listing 4 line#6–7). After that, we invoke the kernel with a specific number of threads and blocks, as well as pass the test case inputs to the kernel (Listing 4 line#8). The kernel executes all functions under test in a GPU and store the results in the AoS of OutputData (Listing 4 line#15,18). Finally, we transfer the output data from the device to the host (Listing 4 line#9).

```
 1 TestDriverParallel(InputData *dataIn, OutputData *dataOut){
 2     int threads = 32
 3     int blocks = n/threads
 4     InputData* dev_in
 5     OutputData* dev_out
 6     cudamemcpy(dev_in,dataIn,sizeof(dataIn), hostTodevice)
 7     cudamemcpy(dev_out,dataOut,sizeof(dataOut), hostTodevice)
 8     kernel<<<blocks, threads>>>(dev_in, dev_out)
 9     cudamemcpy(dataOut,dev_out,sizeof(dataOut), deviceTohost) }
10
11 __global__ void kernel(InputData *dataIn, OutputData *dataOut){
12     int i = blockIdx.x * blockDim.x + threadIdx.x
13     switch(dataIn[i].typeOfSort){
14             case ’b’ :
15             printf("Using bubblesort\n");
16             bubbleSort(dataOut[i].pArrayToSort, dataIn[i].length);
17             break;
18             case ’q’ :
19             printf("Using quicksort\n");
20             quickSort(dataOut[i].pArrayToSort, 0, dataIn[i].length - 1);
21             break;   } }
```

Listing 4: Part of the SortLib test driver pseudocode for parallel version.

Unlike the sequential version, there is no "for loop" in the CUDA version. The kernel will distribute the test case inputs in different threads automatically according to the number of blocks and threads (Listing 4 line#12). The first 32 test case inputs will be assigned to 32 threads in a block. The second 32 test case inputs will be assigned to another 32 threads in a different block and so on. Each thread will have an ID number indicating which test input will be executed. Note that a group of 32 threads will be executed in a warp.

6.5 Profiling

To measure the effectiveness of our approach, we use Nsight Compute CLI which is a profiler tool introduced by NVIDIA [67]. It provides several metrics for the optimization of CUDA applications.

The branch divergence is influenced by the number of inactive threads per warp. To measure the efficiency of a warp, Nsight Compute CLI provides two important metrics (warp execution efficiency and warp non-predicated execution efficiency). In GPUs, many stall reasons cause a warp to be inactive—this is different from active and inactive threads. The most related stall reason to our experiment is stall memory dependency (provided by Nsight Compute CLI) because all selected benchmarks have arrays. The high percentage of stall memory dependency has a negative impact on the performance of a GPU application. Table 5 shows the definition of these metrics based on [68, 69].

6.6 Static code analysis

To collect some statistical data and analyze the code of each benchmark statically, we use Sonargraph-Architect a general-purpose static analysis tool that can

Table 5 Metrics collected from Nsight compute CLI

Nsight compute CLI	Definition
Warp execution efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor
Warp non-predicated execution efficiency	Ratio of the average active threads per warp executing non-predicated instructions to the maximum number of threads per warp supported on a multiprocessor expressed as percentage
Stall memory dependency	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding

Table 6 Metrics used for code analysis and statistical data

Category	Name	Description
Code analysis	Number of duplicated code lines (NDCL)	Number of duplicated lines in duplicated code blocks. The duplicated lines of each code block are calculated as the sum of involved occurrences excluding the largest, which is treated as the reference
	Average block nesting depth (ABND)	Weighted average of nesting depth
Size	Lines of code (LOC)	Lines of code excluding blank and comment lines
	Number of statements (NOS)	Counts all statements
	Source element count (SEC)	Number of programming elements (i.e., types, fields, methods, functions, etc.) plus number of statements
Thomas J.McCabe	Average Complexity (AC)	Weighted average modified cyclomatic complexity

be described as the swiss army knife for architects, quality analysts, and developers [70]. Table 6 lists the metrics that we used. We only consider the measurement that has different values among the selected benchmarks.

7 Results

We execute each test driver of the selected benchmarks more than 20 times and compute the average of the execution times for sequential version with RTI, parallel with RTI, and parallel with ATI. We also run Nsight Compute CLI for parallel versions for the two sets.

We compare RTI with ATI for each selected benchmark by using the Linux command (diff). ATI is not missing a test input when generated by our algorithm (i.e., our algorithm does not delete or add new test inputs to ATI). Both sets have the same test inputs, while they are different in the order of test inputs.

Table 7 Part of BestSet, BadSet, and ATI of Polybench

Test input#	BestSet	BadSet	Arranged test input produced by our algorithm
0	0 434 7	0 434 7	2 151 151
15	0 434 7	9 422 153	2 151 151
31	0 434 7	2 151 151	2 151 151
32	9 422 153	0 434 7	0 434 7
47	9 422 153	9 422 153	0 434 7
63	9 422 153	2 151 151	0 434 7
64	2 151 151	0 434 7	9 422 153
79	2 151 151	9 422 153	9 422 153
95	2 151 151	2 151 151	9 422 153

We validate the output results from the test driver for the two versions (sequential and parallel). The output results are the same for the two versions and the two test inputs sets. However, when a program use floating-point, there are small differences between GPU results and CPU results. A GPU hardware architecture handles floating-point differently from CPU hardware architecture [71].

The floating-point issue appears only in five programs of Polybench. Although IMA has floating-point variables, the processed images from the CPU version and GPU version are identical. Since the other benchmarks deal with integer numbers and strings, the floating-point issue does not impact their output results.

7.1 RQ1: validation and correctness

We use Polybench (the first selected benchmark) and its test inputs to build two sets: (1) best arranged test inputs (BestSet) and (2) badly arranged test inputs (BadSet). BestSet has the same 32 test inputs per warp, so each thread executes the same statement (no branch divergence per warp). BadSet includes different 32 test inputs per warp, so there might be a branch divergence. Then, we apply our algorithm to build ATI from BadSet. After that, we compare ATI with BestSet.

As shown in Table 7, our algorithm generated the expected arranged test set when we compare the generated ATI with BestSet. The algorithm creates a similar set to BestSet such that they have the same 32 test inputs in a group, but the order of these groups is different. If, for example, BestSet has test#1-test#32 in warp#1, our algorithm may put these test inputs in warp#2.

7.2 RQ2: execution time

Table 8 shows the execution time (including data transferring) in seconds for each selected benchmark. The first column is the benchmark name. The second column is the execution time in seconds of a test driver on the CPU version with RTI. The third column is the execution time in seconds of the test driver on the GPU with

Table 8 Execution time of test driver for the six benchmarks in seconds (sec) by using RTI and ATI

Benchmark	CPU time (sec) RTI	GPU time (sec) RTI	GPU time (sec) ATI
IMA	24.5	19.8	6.4
SortLib	56.0	26.5	15.7
Polybench	89.2	55.2	49.1
DynProg	40.3	37.0	49.5
GAlg	195.8	182.3	257.8
EEMBC	0.023	0.007	0.006

Table 9 Data transferring affect the performance of our algorithm

Benchmark	Datatype	#Arguments	#Arrays	Arrays' sizes
IMA	Float, int	11	3	
SortLib	Int	3	1	Different
Polybench	Float, int	12	6	Sizes
DynProg	Int, char	10	6	
GAlgo	Double, int	10	5	
EEMBC	Short, char, int	35	2	Fixed sizes (215)

Table 10 Reasons affect the performance of our algorithm

Benchmark	Warp load imbalance	Cache locality	Control flow
IMA	No	No	If and
SortLib	No	Yes	Switch
EEMBC	No	Yes	Statements
Polybench	No	Yes	
DynProg	Different complexity	Yes	Loops
GAlgo	Different complexity	Yes	

RTI. The last column is the execution time in seconds of a test driver on the GPU version with ATI (generated by our approach).

Running test case inputs on the GPU machine is faster than on the CPU machine for all selected programs with RTI (Table 8 column#2 and #3). When we apply our approach to arrange RTI, the execution time of a test driver for four selected benchmarks is faster than RTI on the GPU machine (Table 8 column#3 and #4).

Although our algorithm arranged the test inputs as excepted, the speedup ratio was affected by a couple of other factors than branch divergence. Table 9 and Table 10 show four reasons for the various speedup achieved between the selected benchmarks: (1) The amount of data transferring between host and device, (2) different control flow paths (if statements and loops), (3) warp load imbalance, and (4) cache locality.

The first factor is data transfer. As shown in Table 9, the datatype, number of arguments, number of arrays, and sizes of arrays are different from one benchmark to another. This provides one explanation of why one benchmark gains a higher speedup ratio than another benchmark. As shown in Fig. 4a, the speedup ratio of the ATI of IMA, SortLib, and EEMBC are 3.8, 3.6, and 3.8, respectively. On the contrary, the speedup ratio of Polybench, DynProg, and GAlgo are 1.8, 0.8, 0.4, respectively. Polybench, DynProg, and GAlgo have at least five arrays transferred from host to device, whereas IMA, SortLib, and EEMBC have at most three arrays transferred from host to device. Therefore, data transfer is one reason that affects the execution time when testing on GPUs.

The second factor is a control flow path of a source code. For example, most of the branches of IMA and SortLib are produced by if statements and switch statements rather than loops (Table 10). Thus, They achieve a better speedup ratio by our algorithm than other benchmarks (Fig. 4a). Although EEMBC has a bigger number of if statements than loops, the difference in the speedup ratio between ATI and RTI is not significant because most of the if statements have one line of code.

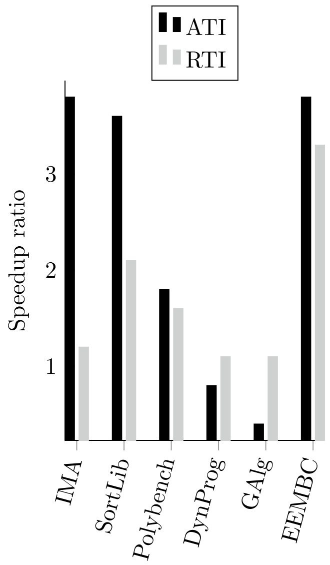
The third factor is warp load imbalance, which occurs when one warp needs time to execute while another warp finishes its execution. Table 8 shows that DynProg and GAlg are slower with the ATI set than the RTI set because they have different functions with different time complexity (Table 10). In DynProg, for example, the time complexity of some of these functions is polynomial (e.g., Fibonacci Numbers). In contrast, the time complexity of some functions is exponential (e.g., Edit Distance and Knapsack). In the GAlg benchmark, Dijkstra's algorithm's time complexity is $O(E \log V)$, whereas the time complexity of Floyd Warshall is $O(V^3)$. This leads to a load imbalance problem. For instance, our algorithm assigns one warp with 32 test inputs invoking the Edit Distance function, while it assigns 32 test inputs calling Fibonacci Numbers function to another warp. With RTI, one warp could have a test input invoking Edit Distance and another test input calling Fibonacci Numbers. Different warps may execute different functions with different execution times (polynomial and exponential). As a result, there is no overhead on one warp performing a function with exponential time in RTI.

With regard to static code analysis, Table 11 shows statistical data results for each benchmark. Our algorithm performs well on the IMA benchmark with the highest LOC, NOS, SEC, and AC, among other benchmarks. Therefore, our algorithm will help speed up the testing execution when a program under test has many LOC, NOS, SEC, and AC.

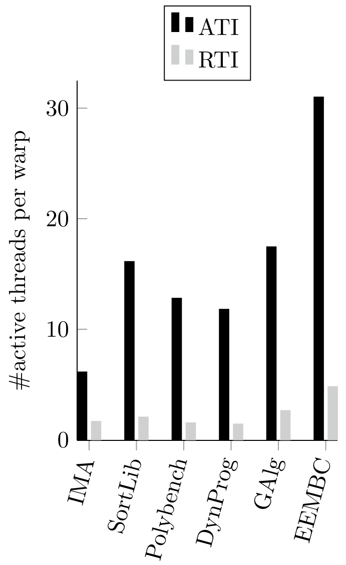
7.3 RQ3: effectiveness

The achieved occupancy of all used benchmarks is 100% for both ATI and RTI. We have 1024 test inputs for a block. A block has up to 32 warps. These 32 warps execute in parallel.

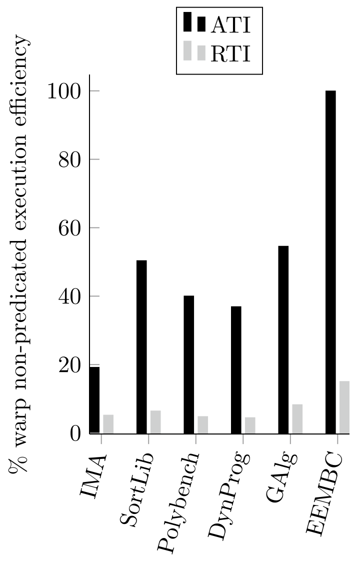
Using ATI generated by our algorithm significantly improves the warp execution efficiency and warp non-predicated execution efficiency for all selected benchmarks (Fig. 4b, c). On the other hand, ATI increases the percentage of the stall memory



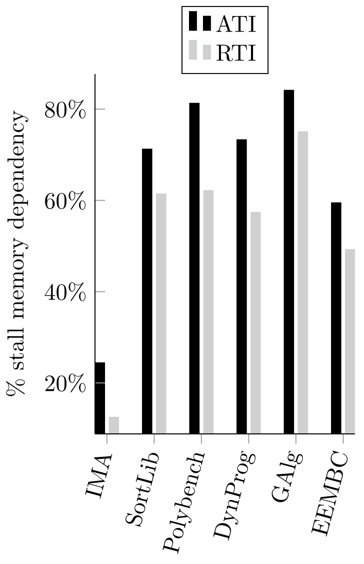
(a) Speedup ratio with respect to the CPU machine.



(b) Warp Execution Efficiency expressed as ratio.



(c) Warp Non-predicated Execution Efficiency expressed as percentage.



(d) Stall Memory Dependency expressed as percentage.

Fig. 4 Speedup ratio and the three Nsight Compute CLI metrics

Table 11 Statistical data of the selected benchmarks gathered by using the metrics described in Table 6

Benchmark	NDCL	ABND	LOC	NOS	SEC	AC
IMA	91	2.91	6,180	4,750	5217	14.86
SortLib	0	2.23	453	375	408	4.58
Polybench	0	2.41	706	530	590	5.14
DynProg	0	2.76	450	358	393	7.51
GAlgo	0	3.3	266	213	243	5.44
EEMBC	119	2.75	938	561	665	12.18

NDCL Number of duplicated code lines, *ABND* average block nesting depth, *LOC* :lines of code, *NOS* Number of statements, *SEC* Source element coun, *AC* Average complexity

dependency (Fig. 4d) for all benchmarks, which negatively impacts the performance of some benchmarks.

7.3.1 Warp execution efficiency

With ATI, there is a significant improvement with respect to the ratio of the number of active threads per warp to the total number of threads per warp in all benchmarks, which ranges between 6.13 and 31.98 threads/warp. RTI has poor warp execution efficiency which ranges between 1.4 and 4.81 threads/warp. As a result, our algorithm reduces the number of inactive threads per warp, which is the main goal of this paper.

The ratio of the active threads per warp is different from one benchmark to another because of two factors. The first factor is the branches produced by loops. Even though our algorithm tries to find similar control flow paths between different test inputs, there might be loops that iterate based on array size. For example, if a test inputs set has 32 test inputs with array size ranges between 2 and 99 and another test inputs with array size ranges between 100 and 500, our algorithm will put the first 32 test inputs in a warp, and the second one in a different warp. The difference in the number of iterations (branches) between threads in the first warp does not exceed 97 iterations. For the second warp, the difference in the number of iterations between threads does not go beyond 400 iterations. This increases the number of branches in one warp versus another warp. Consequently, it reduces the number of active threads per warp. This case appears in all benchmarks except EEMBC. As shown in Table 9, EEMBC has one array with a fixed length (215). This gives a reason why EEMBC achieves the highest ratio of almost 32 threads/warp.

The second factor affecting the warp execution efficiency negatively is branches produced by if statements. If a program under test has a lot of if statements, the warp execution efficiency will decrease. Thus, our algorithm will improve the execution time of this program and the warp execution efficiency. For example, IMA has many if statements in which our algorithm improves its execution time. On the other hand, GAlg benchmark has the fewest number of if statements than the other benchmarks.

Most of its branches are produced by loops. Thus, the execution time of ATI of GAlg is not improved. As a result, our algorithm will be beneficial when a program under test has many if statements.

7.3.2 Warp execution non-predicated efficiency

With ATI, Fig. 4c shows a significant improvement in warp execution non-predicated efficiency which is the percentage of the ratio of active threads executed non-predicated instructions to the total number of executed instructions per warp. Having similar control flow paths of different test inputs increase the number of active threads executing the same instructions at the same time in a warp. For example, if there are 64 test inputs half of them execute branch A and half of them execute branch B, our algorithm will put the 32 test inputs executing branch A in a warp whereas the second 32 test inputs executing branch B will be in another warp. As a result, the number of inactive threads is reduced for non-predicated instructions.

EEMBC has the highest percentage of warp non-predicated execution efficiency. Although it has many if statements, most of them have only one expression and there is no nested if statement. In addition, most of the test inputs provided by the developer for each function under test execute the same if parts. From the generated branch traces (the first step of our algorithm), the distance between one test input to another is zero for 85% of the branches in average. Additionally, EEMBC has loops with the same number of iterations for every test input as the developer specifies the maximum size of the array as a constant. As a result, there is not a huge difference between RTI and ATI for EEMBC in terms of their execution times.

For the other benchmarks, the presence of loops that produce branches affects the warp non-predicated efficiency as each test input iterates with a different number of iterations as we discussed previously.

7.3.3 Stall memory dependency

Since all benchmarks have arrays, the most stall reasons could be related to memory dependency. Array size affects the memory access pattern (cache locality) in a GPU machine because the amount of cache is smaller than in regular CPUs. Figure 4d shows that five benchmarks have more than 50% stall memory dependency. The percentage increases with a generated arranged set because the algorithm will put test inputs with a large array size (e.g., 500–550) in one warp whereas it will put test inputs with small array size (e.g., 10–50) in another warp. Thus, the memory access pattern and memory coalescing will affect the first warp since all test inputs in this warp have a bigger array size than the second set of test inputs.

The lowest stall memory dependency is for IMA. Compared with the other benchmarks, IMA has the most number of LOC (Table 11) making stall instruction fetch to be 10%. Other benchmarks have less than 1% stall instruction fetch. Also, the array size of the majority of IMA is less than the array size in the other benchmarks.

8 Discussions

To make an adequate unit test to catch bugs, we may invoke a function thousands of times with different inputs. The different test inputs are considered representative tests and relevant to a program under test. The restriction of 32 threads/warp in a GPU machine provides an opportunity to give each function under test batches of 32 test inputs.

The time complexity of our algorithm is a polynomial time in terms of number of test inputs (n) and number of branches (b). For the traces generator step, the time complexity is $O(nb)$. The time complexity of the similarity matrix step is $O(n^2)$. In a complete graph, the number of edges (e) is as following: $e = \frac{n(n-1)}{2}$. The time complexity of buckets constructor step has the following: 1) for MST $O(e \log n)$, 2) for connected components $O(n + e)$, and 3) for reading data from buckets and storing in ATI $O(n)$. As a result, the overall time complexity of our algorithm is dominated by $O(n^2 \log n)$, which is scalable.

One may argue that our parallelization may introduce "test-order dependencies" in which one iteration may impact values used by other iterations. We parallelize the test execution in an isolation manner such that each thread has its input data (no shared data). If a function needs to be executed after another function, we test them in the same order.

The limitations of CUDA (e.g., not supporting String and read file) did not allow us to run our experiment on real large-scale applications. We believe these limitations could be resolved in the future by adding the C libraries for CUDA applications.

Our proposed approach could be applied to multiple program languages. For example, a test suite could be implemented by using CUDA Python to test Python code and CUDA Jave to test Java code.

The proposed algorithm could be improved to consider the warp imbalance problem (e.g., heavy tasks will be distributed to different warps instead of one warp). The algorithm could be further improved to determine whether it should distribute test inputs with branch divergence or warp load imbalance depending on the control flow path under test. Since our algorithm is related to the clustering problem, the algorithm might be further improved for the clone detection problem.

In our experiment, we use NVIDIA Volta architecture Tesla V100 that supports the independent thread scheduling feature. This feature determines how to group active threads from the same warp together. Threads can diverge and reconverge at sub-warp granularity, while the convergence optimizer in Volta will still group together threads which are executing the same code and run them in parallel for maximum efficiency. Note that execution is still SIMT, retaining the execution efficiency of previous architectures [72]. As a result, the speedup ratios (Fig. 4a) are not significant although our algorithm reduces the number of inactive threads per warp (Fig. 4b, c).

All the implementations and instructions of how to apply our algorithm for practitioners are available on our GitHub repository text.[2]

9 Threats to validity

9.1 Conclusion validity

Our experiment has a small number of benchmarks, which leads to low statistical power and tests. Due to the limitation of CUDA, which does not support some of C standard libraries such as String, we were not able to use any arbitrary benchmarks. For example, we could not test programs in the SIR-C [73] benchmark since all of these programs use a C standard library and read and write to a file. We believe this limitation can be resolved in the future by implementing these C standard libraries for CUDA programs.

9.2 Internal validity

Although we use only six benchmarks, they vary in different ways. We use EEMBC benchmark used by [1, 15]. We use Polybench, which is widely used in high-performance computing communities. We include four benchmarks from GitHub to add other types of program structure different from EEMBC and Polybench. We search for source code that is well commented and easy to understand.

9.3 Construct validity

In our experiments, we use AoS and CUDA similar to [15]. In [74], they designed the input and output data as AoS and showed this design was better than Struct of Arrays and parallel arrays of inputs and outputs. Also, they showed that CUDA is better than Open-MP Offloading.

9.4 External validity

We could not study our algorithm with C real large-scale applications since all C real-world applications use C standard libraries. This limits the ability to generalize the results beyond the experiment setting. For example, in IMA (a real small-scale application in C), we only test functions that do not use unsupported C standard libraries by CUDA.

[2] https://github.com/tbagies/GPU-BranchDivergence.

Some selected benchmarks do not have test inputs provided by developers (e.g., GAlg [65]). We generate their test inputs automatically such that test inputs cover all functions as well as different cases. For example, we generate different graphs such as an undirected graph, directed graph, dense graph and complete graph for GAlg.

9.5 Portability validity

We did our algorithm on a specific platform (NVIDIA). We specified the warp size and number of threads per warp with respect to NVIDIA specification. If you move to different architecture, you may need some fine-tuning based on the chosen architecture specifications, such as changing the warp size.

9.6 Reproducibility validity

Our algorithm should not be repeated and reproduce new traces or a similarity matrix whenever a program under test changes. If, for example, we have already run the testing with an ATI set generated from our algorithm based on traces and similarity matrix of an old version of the program under test. Then, the program changes and has a new version. The old ATI set may not work well as it is built based on the old version. In other words, the generated ATI set might need to be rearranged. However, we do not want to repeat the whole execution of our algorithm (e.g., collecting branch traces and producing a new similarity matrix). Instead, we could use some regression testing techniques [75, 76] in which we eliminate the test inputs that were not affected by the change. The remaining test inputs that were affected by the change could be rearranged based on a new step that should be added to our algorithm (we consider this as future work).

10 Conclusion and future work

When parallelizing test execution on a GPU, each test input is executed by a thread and may have a different control flow path, which leads to divergent instructions between threads. Some threads are inactive, waiting for other threads to finish their execution. Therefore, the branch divergence among threads in a warp increases the overall test execution time. We propose an algorithm that arranges the test inputs concerning their control flow path to reduce the branch divergence when executing tests on GPUs.

Our approach helps in grouping similar control flow paths of test case inputs to be executed in 32 threads per warp. It shows that arranging the test case inputs yields faster execution time of four of six selected benchmarks. Also, it improves the warp execution efficiency on the GPU machine for all tested benchmarks.

Our approach is the first step to build a set of ATI. A direction for future work would be utilizing some regression testing techniques to re-arrange test inputs that are affected by adding a new function or deleting an existing function. In addition, we would address the load imbalance problem when a program under test has different functions with different time complexities (exponential, polynomial), or has test inputs with small and large array sizes.

Acknowledgements We would like to thank King Abdulaziz University, Jeddah, Saudi Arabia for supporting the Ph.D. scholarship to work on research.

Author contributions TB wrote the main manuscript text and prepared all figures and tables. AJ reviewed the manuscript

Funding The authors declare no funding.

Declarations

Conflict of interest The authors declare that there is no conflict of interest.

References

1. Yaneva V, Rajan A, Dubach C (2017) Compiler-assisted test acceleration on gpus for embedded software. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2017, pp. 35–45. ACM, New York, NY, USA. https://doi.org/10.1145/3092703.3092720
2. Harrold MJ (2000) Testing: A roadmap. In: Proceedings of the Conference on The Future of Software Engineering. ICSE ’00, pp. 61–72. ACM, New York, NY, USA. https://doi.org/10.1145/336512.336532
3. Shete N, Jadhav A (2014) An empirical study of test cases in software testing. In: International Conference on Information Communication and Embedded Systems (ICICES2014), pp. 1–5. https://doi.org/10.1109/ICICES.2014.7033883
4. Sommerville I (2015) Software Engineering, 10th edn. Pearson, ???
5. Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Trans Software Eng 27(10):929–948. https://doi.org/10.1109/32.962562
6. Gambi A, Kappler S, Lampel J, Zeller A (2017) Cut: Automatic unit testing in the cloud. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2017, pp. 364–367. ACM, New York, NY, USA
7. Kappler S (2016) Finding and breaking test dependencies to speed up test execution. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, pp. 1136–1138. ACM, New York, NY, USA. https://doi.org/10.1145/2950290.2983974
8. Liu C-H, Chen S-L, Chen W-K (2017) Cost-benefit evaluation on parallel execution for improving test efficiency over cloud. In: 2017 International Conference on Applied System Innovation (ICASI), pp. 199–202. https://doi.org/10.1109/ICASI.2017.7988384
9. Oriol M, Ullah F (2010) Yeti on the cloud. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, pp. 434–437. https://doi.org/10.1109/ICSTW.2010.68
10. Parveen T, Tilley S, Daley N, Morales P (2009) Towards a distributed execution framework for junit test cases. In: 2009 IEEE International Conference on Software Maintenance, pp. 425–428. https://doi.org/10.1109/ICSM.2009.5306292
11. Gambi A, Gorla A, Zeller A (2017) O!snap: Cost-efficient testing in the cloud. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 454–459. https://doi.org/10.1109/ICST.2017.51

12. von Hof V, Fuchs A (2018) Automatic scalable parallel test case execution. introducing the münster distributed test case runner for java (midstr). In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 1062–1064
13. Koong C-S, Shih C-H, Wu C-C, Hsiung P-A (2013) The architecture of parallelized cloud-based automatic testing system. In: 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems, pp. 467–470. https://doi.org/10.1109/CISIS.2013.85
14. Duarte A, Cirne W, Brasileiro F, Machado P (2006) Gridunit: software testing on the grid. In: Proceedings of the 28th International Conference on Software Engineering, pp. 779–782
15. Rajan A, Sharma S, Schrammel P, Kroening D (2014) Accelerated test execution using gpus. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ASE '14, pp. 97–102. ACM, New York, NY, USA. https://doi.org/10.1145/2642937.2642957
16. Han TD, Abdelrahman TS (2011) Reducing branch divergence in gpu programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. GPGPU-4, pp. 3–138. ACM, New York, NY, USA. https://doi.org/10.1145/1964179.1964184
17. Zhang EZ, Jiang Y, Guo Z, Shen X (2010) Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In: Proceedings of the 24th ACM International Conference on Supercomputing. ICS '10, pp. 115–126. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1810085.1810104
18. Yu Z, Eeckhout L, Xu C (2016) Thread similarity matrix: Visualizing branch divergence in gpgpu programs. In: 2016 45th International Conference on Parallel Processing (ICPP), pp. 179–184
19. Coutinho B, Sampaio D, Pereira FMQ, Meira Jr W (2011) Divergence analysis and optimizations. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp. 320–329. IEEE
20. Sampaio D, Martins R, Collange S, Pereira FMQ (2012) Divergence analysis with affine constraints. In: 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing, pp. 67–74
21. Kerr A, Diamos G, Yalamanchili S (2009) A characterization and analysis of ptx kernels, pp. 3–12. https://doi.org/10.1109/IISWC.2009.5306801
22. Sartori J, Kumar R (2013) Branch and data herding: reducing control and memory divergence for error-tolerant gpu applications. IEEE Trans Multimedia 15(2):279–290
23. Vespa LVL (2018) Unraveling the divergence of gpu threads. In: 2018 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 1398–1403
24. Chakroun I, Mezmaz M, Melab N, Bendjoudi A (2013) Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm. Concurr Comput Pract Exp 25(8):1121–1136
25. Li Y, Liu R (2016) High throughput gpu polar decoder. In: 2016 2nd IEEE International Conference on Computer and Communications (ICCC), pp. 1123–1127
26. Carrillo S, Siegel J, Li X (2009) A control-structure splitting optimization for gpgpu. In: Proceedings of the 6th ACM Conference on Computing Frontiers, pp. 147–150
27. Reissmann N, Falch TL, Bjørnseth BA, Bahmann H, Meyer JC, Jahre M (2016) Efficient control flow restructuring for gpus. In: 2016 International Conference on High Performance Computing Simulation (HPCS), pp. 48–57
28. Anantpur J, Govindarajan R (2014) Taming control divergence in gpus through control flow linearization. In: International Conference on Compiler Construction, pp. 133–153. Springer
29. Zone ND (2021) CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture. Accessed on 23 June 2021
30. Gupta P (2020) CUDA Refresher: The CUDA Programming Model. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/. Accessed on 27 June 2021
31. Han TD, Abdelrahman TS (2011) hicuda: High-level gpgpu programming. IEEE Trans Parallel Distrib Syst 22(1):78–90. https://doi.org/10.1109/TPDS.2010.62
32. Lin Y, Grover V (2012) Using CUDA warp-level primitives. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/. Accessed on 14 Oct 2018
33. Workshop V (2019) Introduction to GPGPU and CUDA programming: thread divergence. https://cvw.cac.cornell.edu/gpu/thread_div. Accessed on 20 Aug 2019
34. Srivastava A, Thiagarajan J (2002) Effectively prioritizing tests in development environment. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 97–106

35. Wong WE, Horgan JR, London S, Agrawal H (1997) A study of effective regression testing in practice. In: Proceedings The Eighth International Symposium on Software Reliability Engineering. pp 264–274
36. Beller M, Gousios G, Panichella A, Zaidman A (2015) When, how, and why developers (do not) test in their ides. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, pp. 179–190. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2786805.2786843
37. Rothermel G, Untch RH, Chu C, Harrold MJ (1999) Test case prioritization: An empirical study. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360), pp. 179–188. IEEE
38. Zhang S, Jalali D, Wuttke J, Muşlu K, Lam W, Ernst MD, Notkin D (2014) Empirically revisiting the test independence assumption. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 385–396
39. Lam W, Zhang S, Ernst MD (2015) When tests collide: evaluating and coping with the impact of test dependence. University of Washington Department of Computer Science and Engineering, Tech, Rep
40. Schwahn O, Coppik N, Winter S, Suri N (2019) Assessing the state and improving the art of parallel testing for c. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 123–133
41. Hu H, Jiang C-H, Ye F, Cai K-Y, Huang D, Yau SS (2010) A parallel implementation strategy of adaptive testing. In: 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, pp. 214–219. https://doi.org/10.1109/COMPSACW.2010.44
42. Misailovic S, Milicevic A, Petrovic N, Khurshid S, Marinov D (2007) Parallel test generation and execution with korat. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 135–144
43. Siddiqui JH, Khurshid S (2009) Pkorat: Parallel generation of structurally complex test inputs. In: 2009 International Conference on Software Testing Verification and Validation, pp. 250–259. https://doi.org/10.1109/ICST.2009.48
44. Fung WWL, Sham I, Yuan G, Aamodt TM (2007) Dynamic warp formation and scheduling for efficient gpu control flow. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pp. 407–420
45. Brunie N, Collange S, Diamos G (2012) Simultaneous branch and warp interweaving for sustained gpu performance. In: 2012 39th Annual International Symposium on Computer Architecture (ISCA), pp. 49–60
46. Rhu M, Erez M (2012) Capri: prediction of compaction-adequacy for handling control-divergence in gpgpu architectures. ACM SIGARCH Comput Arch News 40(3):61–71
47. Rhu M, Erez M (2013) Maximizing simd resource utilization in gpgpus with simd lane permutation. In: Proceedings of the 40th Annual International Symposium on Computer Architecture. pp. 356–367
48. Fung WWL, Aamodt TM (2011) Thread block compaction for efficient simt control flow. In: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture. HPCA '11, pp. 25–36. IEEE Computer Society, USA
49. Li B, Wei J, Guo W, Sun J (2015) Improving simd utilization with thread-lane shuffled compaction in gpgpu. Chin J Electron 24:684–688. https://doi.org/10.1049/cje.2015.10.004
50. Yang H, Chen S, Wan J, Xu X (2015) Divergent branch threads compaction for efficient simd control flow. Chin J Electron 24(2):288–294
51. Narasiman V, Shebanow M, Lee CJ, Miftakhutdinov R, Mutlu O, Patt YN (2011) Improving gpu performance via large warps and two-level warp scheduling. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-44, pp. 308–317. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2155620.2155656
52. Meng J, Tarjan D, Skadron K (2010) Dynamic warp subdivision for integrated branch and memory divergence tolerance. In: Proceedings of the 37th Annual International Symposium on Computer Architecture. pp. 235–246
53. Tarjan D, Meng J, Skadron K (2009) Increasing memory miss tolerance for simd cores. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 1–11

54. Emmery C (2017) Euclidean vs. cosine distance. https://cmry.github.io/notes/euclidean-v-cosine. Accessed on 27 June 2021
55. Ladd JR (2020) Understanding and using common similarity measures for text analysis. https://programminghistorian.org/en/lessons/common-similarity-measures. Accessed on 11 Jul 2021
56. Nvidia (2018) cuda-c-programming-guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability. Accessed on 28 Jul 2018
57. die.net (2011) clock_gettime(3); Linux man page. https://linux.die.net/man/3/clock_gettime. Accessed on 16 Oct 2018
58. Yang C-T, Huang C-L, Lin C-F (2011) Hybrid cuda, openmp and mpi parallel programming on multicore gpu clusters. Comput Phys Commun. 182(1):266–269. https://doi.org/10.1016/j.cpc.2010.06.035. Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15–19, 2009
59. Harris M (2011) How to Implement Performance Metrics in CUDA C/C++. https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/. Accessed on 16 Oct 2018
60. NVIDIA (2012) NVIDIA CUDA C Programming Guide. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessed on 26 Jan 2019
61. Pouchet LN (2016) PolyBench/C 3.2. http://polybench.sourceforge.net. Accessed on 14 Oct 2018
62. Kalliamvakou E, Damian D, Blincoe K, Singer L, German DM (2015) Open source-style collaborative development practices in commercial projects using github. In: Proceedings of the 37th International Conference on Software Engineering: Volume 1. ICSE '15, pp. 574–585. IEEE Press, Piscataway, NJ, USA. http://dl.acm.org/citation.cfm?id=2818754.2818825
63. Quijada M (2014) image-manipulation-in-c. https://github.com/mauryquijada/image-manipulation-in-c.git. Accessed on 29 Jul 2018
64. Yerburgh E (2017) c-sorting-algorithms. https://github.com/eddyerburgh/c-sorting-algorithms/tree/master/algorithms. Accessed on 17 Aug 2019
65. Felipe L (2018) VAR-solutions. https://github.com/luizok/GraphAlgorithms/blob/master/graphalgs.c. Accessed on 14 Jan 2020
66. Varshney R (2018) VAR-solutions. https://github.com/VAR-solutions/Algorithms/tree/dev/Dynamic%20Programming. Accessed on 14 Jan 2020
67. Corporation N (2019) Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html. Accessed on 4 Sep 2019
68. NVIDIA (2018) NVIDIACUDA Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference. Accessed on 14 Oct 2018
69. Corporation N (2019) Nsight Compute CLI-5.3. Metric Comparison. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#nvprof-metric-comparison. Accessed on 4 Sept 2019
70. Family SP (2021) Sonargraph-Architect. https://www.hello2morrow.com/products/sonargraph/architect9. Accessed on 16 Nov 2021
71. Whitehead N, Fit-Florea A (2011) Precision & performance: floating point and ieee 754 compliance for nvidia gpus. rn (A+ B) 21(1):18749–19424
72. NVIDIA (2017) NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed on 28 Apr 2023
73. Repository S-aI: SIR Usage Information. https://sir.csc.ncsu.edu/portal/usage.php. Accessed on 3 Apr 2019
74. Bagies T, Jannesari A (2021) An empirical study of parallelizing test execution using cuda unified memory and openmp gpu offloading. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 271–278. https://doi.org/10.1109/ICSTW52544.2021.00052
75. Zhang L (2018) Hybrid regression test selection. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 199–209
76. Marijan D, Liaaen M (2018) Practical selective regression testing with effective redundancy in interleaved tests. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 153–162