Measurement Based Execution Time Analysis of GPGPU Programs via SE+GA

Adrian Horga
Linköping University
Sweden
adrian.horga@liu.se

Sudipta Chattopadhyay
Singapore Univ. of Tech. and Design
Singapore
sudipta_chattopadhyay@sutd.edu.sg

Petru Eles
Linköping University
Sweden
petru.eles@liu.se

Zebo Peng
Linköping University
Sweden
zebo.peng@liu.se

***Abstract*—Understanding the execution time is critical for embedded, real-time applications. Worst-case execution time (WCET) is an important metric to check the real-time constraints imposed on embedded applications. For complex execution platforms, such as graphics processing units (GPUs), analysis of WCET imposes great challenges due to the complex characteristics of GPU architecture as well as GPU program semantics. In this paper, we propose `GDivAn`, a measurement-based WCET analysis tool for arbitrary GPU kernels. `GDivAn` systematically combines the strength of symbolic execution (SE) and genetic algorithm (GA) to maintain both the scalability and the effectiveness of the analysis process. Our evaluation with several open-source GPU kernels reveals the efficiency of `GDivAn`.**

***Index Terms*—GPU, symbolic execution, genetic algorithm, testing**

I. Introduction

General-purpose graphics processing units (GPGPU) have been gaining attention due to their high computational capacity and low power consumption. It is, therefore, natural to investigate the capability of GPGPUs in real-time applications [1], [2]. This includes applications in avionics and automotive, among others. To streamline the adoption of GPGPUs in real-time processing, a major research thrust is to enable reasoning over the timing behaviour of GPGPU applications.

The analysis of worst-case execution time (WCET) is crucial to check whether a given real-time application meets the deadline. WCET captures the maximum execution time for a given program over all its inputs. For a given GPGPU program, its WCET depends on the program and its features as well as the GPU and its features. Therefore, in principle, it is possible to determine the WCET for a given GPGPU program running on a given GPU-based platform. In this paper, we propose `GDivAn`, an approach to systematically generate test inputs with the aim of exposing the WCET of an arbitrary GPGPU program. `GDivAn` complements measurement-based timing analysis methodologies, specifically, in terms of obtaining effective test inputs that are likely to lead to WCET measurements.

The design and implementation of `GDivAn` involves several technical challenges. Firstly, GPGPU programs employ massive parallel processing. Hence, it is infeasible to generate all possible execution scenarios in realistic GPGPU programs. To this end, we first symbolically execute a GPGPU program with a small number of threads. We employ such a strategy to hypothesize that it is often possible to cover the GPGPU program features (e.g. all possible branch outcomes) with a small number of threads. Secondly, GPUs involve complex micro-architectural features, involving shared memory, hundreds of processing units and caches. Moreover, the exact nature of these micro-architectural features (e.g. cache replacement policy and bus arbitration policy) remain opaque to developers. This, in turn, makes the works on static WCET analysis [3], [4], [5] practically infeasible for GPGPU programs. To solve such challenges, we directly execute the generated test inputs in commodity GPU-based systems. Thus, the key novelty in our approach is to drive the generation of such test inputs to expose the WCET. To this end, we leverage the results obtained from symbolically executing a GPGPU program with a small number of threads. Subsequently, we complement these results via a genetic algorithm to systematically search the input space and potentially converge towards the WCET.

How does `GDivAn` differ from the state-of-the-art? Existing works in timing analysis of GPGPU applications [1], [6] consider the system-level schedulability analysis assuming the WCET of individual programs as given and ignoring how it can be produced, whereas we focus on the structure of individual GPGPU programs and produce their WCET. In the past few years, there has been a rise on using extreme value theory (EVT) for measurement-based timing analysis [7], [8], [9]. However, there are strong requirements to be satisfied for any result of an EVT-based approach to produce a sound WCET [9]. Such is the collection of a representative input sample that needs to satisfy key assumptions like independence and identical distribution. While techniques have been proposed to achieve some of these requirements, they need deep interventions in the hardware/software architecture [9] and are not applicable to processors of the complexity typical to a GPU. In general, even in the case of less complex, classical architectures there are several open challenges to be solved before EVT-based techniques can be generally applied and trusted [10], [11], [12]. In general, existing works on the WCET analysis for GPUs [7], [8], [13] ignore the systematic generation of test inputs for exposing the WCET. In addition, hybrid WCET analysis [13] is not applicable for commodity GPU-based systems. This is due to its implicit assumption

on the availability of GPU execution model. Finally, GDivAn sets itself apart from existing works on combining symbolic execution and genetic algorithm [14], by its novel mechanism for testing the WCET of GPGPU programs.

After providing an overview in Section III, we make the following contributions in the paper.

1) We propose GDivAn, a novel approach that employs a synergistic combination of symbolic execution (SE) and genetic algorithm (GA). This is to analyze the timing properties of arbitrary GPGPU programs (Section IV).
2) We implement GDivAn for commodity GPU hardware (i.e. NVIDIA Tegra K1 GPU). Such an implementation can easily be integrated with any measurement-based timing analysis for GPGPU programs.
3) We evaluate GDivAn for several GPU kernels involving up to tens of thousands of threads (Section V). Our evaluation reveals that GDivAn is significantly more effective in exposing the WCET compared to both random testing and genetic algorithm in isolation. Our implementation and all experimental data are publicly available.

II. System and Execution Model

In this paper, we target GPU kernels written in CUDA [15] [1] and execution platforms similar to NVIDIA GPUs. However, we believe that the core scientific capabilities within GDivAn are also applicable to other GPU platforms. The smallest execution unit in CUDA programs is a *thread* and the program running on the GPU is called a *kernel*. Several threads can be grouped into a *thread block*. GPUs use Streaming Multiprocessors (SMs) to execute the kernels assigned to them. Each SM has its own memory subsystem. Such a memory subsystem typically involves registers, scratchpad memories and multiple levels of caches. All the SMs have access to the global memory and all the threads within a thread block run on the same SM. SMs leverage the single-instruction-multiple-threads (SIMT) paradigm to employ large-scale parallelism. To this end, threads in a thread block are grouped into *warps*. All the threads within a warp execute instructions in lock-step.

Typically SMs in a GPU do not employ branch prediction. If two threads of the *same warp* activate different targets of a branch instruction, then a phenomenon, commonly known as *branch divergence*, takes place. Branch divergence may significantly affect the level of parallelism offered by GPUs. For a typical "`if (C) then A else B`" structure, branch divergence leads to a serial executions of `A` and `B`. Threads that do not activate the *true* leg of conditional `C` are disabled while `A` is executed. Likewise, all threads satisfying the conditional `C` are disabled while `B` is executed.

III. Overview

In this section, we discuss the challenges in estimating WCET of GPU-based programs via simple examples. Subsequently, we show the key insight behind our approach.

[1] GDivAn is equally applicable to other GPU programming paradigms like OpenCL

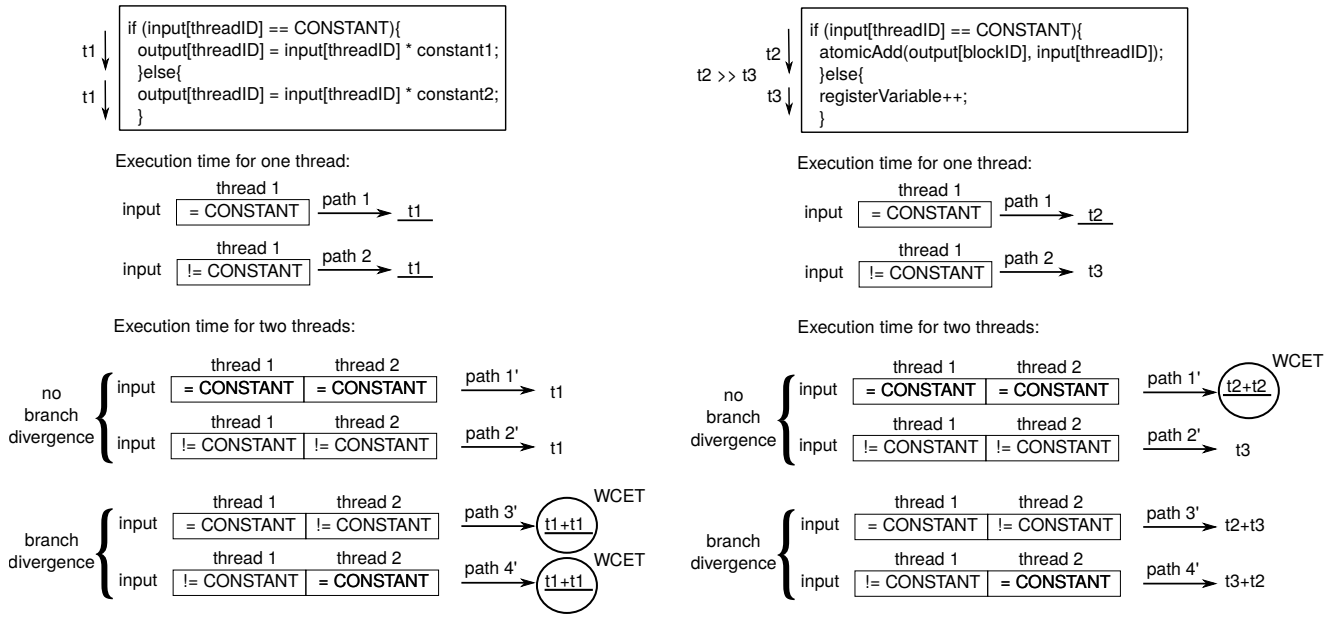
Challenges in WCET analysis for GPU-based programs. Consider the GPU kernels shown in Figure 1. For the sake of simplicity in this example, we assume that any pair of threads, executing the same instruction in the kernels, are free from memory contention. In Figure 1(a), let us assume that the multiplication instruction takes time $t1$ to execute. Hence, for each thread, the worst-case execution time (WCET) is $t1$. However, GPU kernels typically involve thousands of threads running in parallel. If the kernel in Figure 1(a) is executed for two threads, then it leads to four different execution scenarios depending on the value of `input`. As shown in Figure 1(a), the WCET (*i.e.* $t1+t1$) is manifested for $path\ 3'$ and $path\ 4'$. This is due to the divergence that takes place at the branch instruction.

From the discussion in the preceding paragraph, we may hypothesize that branch divergence always leads to longer execution time. The example in Figure 1(b) contradicts this hypothesis. In Figure 1(b), the true leg of the branch involves an atomic add operation, which, in turn might access the slow global-memory. In contrast, the false leg of the branch involves simple arithmetic manipulations on registers. Accessing global-memory is several orders of magnitudes slower than accessing register variables. Hence, in Figure 1(b), $t2 \gg t3$, where $t2$ ($t3$) is the time to execute the true (false) leg of the branch. If two threads execute the kernel in Figure 1(b), then the WCET is $t2 + t2$, due to the atomic nature of the operation which imposes serialization. We note that the WCET is manifested for an execution scenario that does not exhibit branch divergence. Intuitively, this occurs due to the unbalanced execution time across different legs of the branch instruction.

Will random testing work? In Figure 1, the number of execution scenarios grows exponentially with the number of threads. In particular, consider to use random testing for exposing WCETs of the examples in Figure 1. We observe that random testing only has a slim chance (probability $< 0.4\%$ for two threads) to synthesize input vectors equal to $CONSTANT$. Since the WCET is manifested only for such input vectors, it is unlikely that random testing will converge towards exposing the WCET of programs in Figure 1.

Will symbolic execution work? Symbolic execution poses an attractive choice to systematically explore all unique execution paths in an application. It leverages the power of constraint solvers to symbolically capture all inputs exhibiting an execution path. Then, such a symbolic formula is manipulated to generate inputs for a different execution path. As a result, if our example programs in Figure 1 are executed with two threads, symbolic execution terminates generating four test inputs – one each for a unique execution scenario. Thus, the probability to expose WCET in our example program increases to 100% if all four symbolically detected paths are executed.

Unfortunately, the complexity of symbolic execution quickly becomes intractable with the growing number of threads. As GPUs are targeted to support massive multi-processing, typically GPGPU programs involve thousands of



(a) Balanced branch

(b) Unbalanced branch

Fig. 1. **Motivational example**. $thread\ 1$ and $thread\ 2$ belong to the same warp. Both examples use one input and one output vector. Both vectors are stored in GPU global-memory. The variable $threadID$ captures the global identity of a thread within the execution. For each thread, variable $blockID$ captures the identity of the thread block. Threads in a warp belong to the same thread block.

threads. As a result, it is infeasible to explore all possible execution paths (via symbolic execution) for any realistic GPGPU applications.

Key insight. Our `GDivAn` approach proposes a novel mechanism to circumvent the inherent complexity of symbolic execution, yet use its power to cover the structure of GPGPU programs. Our key intuition is to run a GPU kernel symbolically only for a limited number of threads and to cover all (or most) execution paths of this kernel. Subsequently, we investigate each path with respect to some program features (e.g. number of instructions) that influence timing. Leveraging an SMT solver, we generate test inputs for each path. Finally, we systematically scale and manipulate these test inputs for the original kernel that potentially runs with significantly larger number of threads. The final stage involves a novel application of genetic algorithm to manipulate the test inputs. In essence, our `GDivAn` approach combines the strength of symbolic execution to explore program structure and the strength of genetic algorithm to systematically search a large input space.

How `GDivAn` works. Figure 2 provides an outline of `GDivAn`. At a high level, the workflow of `GDivAn` involves three steps: 1) generation of input atoms, 2) scaling of atoms, and 3) exploration of input space via genetic algorithm.

1) Generation of input atoms: Consider the GPU kernel shown in Figure 1(a). We use the predicate $pred_e(th)$ to symbolically capture the execution of control flow edge e in thread th. Concretely, $pred_e(th)$ is *true* if control flow edge e is executed in thread th. Otherwise, $pred_e(th)$ is set to *false*. Let us assume that the *true* and *false* legs of the conditional in

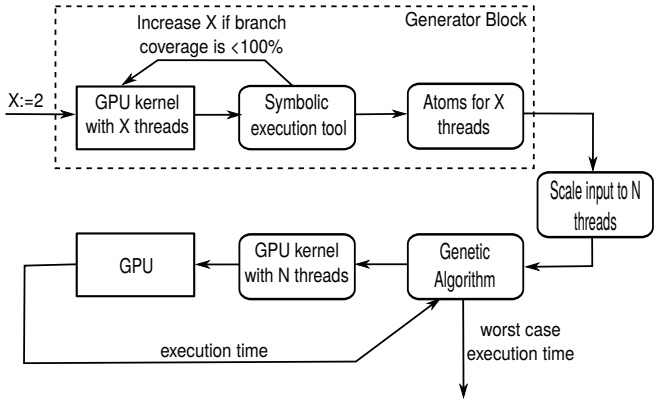


Fig. 2. Overview of estimation tool

Figure 1(a) capture control flow edges $e1$ and $e2$, respectively. In order to generate input atoms, we symbolically execute a given GPU kernel with a small number of threads. For instance, using two threads (say, thread 0 and thread 1), a symbolic execution of the code in Figure 1(a) will result in the following set of execution paths: *(a)* $path_1 : pred_{e1}(0) \wedge pred_{e1}(1)$, *(b)* $path_2 : pred_{e2}(0) \wedge pred_{e2}(1)$, *(c)* $path_3 : pred_{e1}(0) \wedge pred_{e2}(1)$, and *(d)* $path_4 : pred_{e2}(0) \wedge pred_{e1}(1)$. We note that $path_{1...4}$ symbolically captures all inputs leading to the respective execution paths. We leverage on constraint solvers to generate input atoms from these symbolic formulas. The primary goal of this step is to explore the GPU-kernel structure for a small number of threads (i.e. two threads in

this example) and generate representative test inputs covering the structure of the GPU-kernel. Our intuition is that it is often feasible to cover the structure of GPU kernels (e.g. all branches in the kernel) despite being executed with a small number of threads.

2) Scaling input atoms: Input atoms, as discussed in the preceding paragraphs, can be used to execute the given GPU kernel with a small number of threads. To obtain an initial set of test inputs for the original kernel, which typically executes with thousands of threads, we systematically scale the input atoms obtained via symbolic execution. Figure 3 outlines input atoms generated from symbolically executing the code (also shown in Figure 3) with two threads. Subsequently, these input atoms were scaled, as shown on the top right corner of Figure 3.

3) Exploration of input space via genetic algorithm: Steps 1) and 2) generate test inputs that reduce the search space. However, the paths that heavily affect the execution times through branch divergence, instruction serialization or cache contention need to be detected from the large remaining search space. We leverage genetic algorithm (GA) to systematically explore this search space. To this end, we first run the given GPU kernel with the set of scaled inputs (obtained from the previous stage) and obtain the execution time for each such input. Based on the execution times obtained, GA builds new inputs via a series of selection, crossover and mutation operation on the current set of test population and input atoms obtained via the symbolic execution. Our objective in the GA is to maximize the execution time of the GPU kernel. Figure 3 contains an example of two new input vectors. These input vectors were created using two of the current input vectors and input atoms, as shown in Figure 3. The process of executing test inputs and generating new test population via GA is repeated until the execution time does not show significant variation across two consecutive generations of GA.

IV. Detailed Methodologies

In this section, we describe the mechanism of different building blocks in `GDivAn`. The overall outline of `GDivAn` and the inter-dependencies between its building blocks appear in Figure 2.

A. Generator block

The purpose of this block is to enable the creation of an initial test population. To this end, we generate test inputs via symbolic execution. The test inputs are generated to cover the structure (e.g. the branches) of a GPU kernel. As discussed in Section III, it is practically infeasible to employ symbolic execution for realistic GPU kernels running a large number of threads. Hence, we apply symbolic execution in a trimmed down version of a given GPU kernel. Such a trimming is employed by symbolically executing the GPU kernel only with a small number of threads and aiming to obtain *branch coverage*. We describe this in the following.

Trimmed symbolic execution: We note that it requires at least two threads to manifest branch divergence in a GPU kernel (see Figure 1). In Figure 1(a), we observed that the presence of branch divergence may lead to longer execution time. The primary intuition behind obtaining the branch coverage is to have representative inputs in the test population that trigger branch divergence. To employ our trimmed version of symbolic execution, we systematically increase the number of threads (starting from two threads) in the GPU kernel and invoke the symbolic execution engine. For each invocation of the symbolic engine, we measure the branch coverage being obtained. Finally, we stop the symbolic execution process once both legs of all branch instructions are covered. However, for complex GPU kernels, it might even be infeasible to obtain 100% branch coverage within a reasonable time. For such cases, we impose a time bound (<1 hour) on the symbolic runs of the kernel.

Let us assume that symbolic execution of the kernel was performed for X number of GPU threads. Upon termination of the symbolic execution, it generates the set of all execution paths in the kernel trimmed down to X threads. For each explored path π, we collect the following information:

$$P_\pi(X) \equiv \langle form_\pi, brdiv_\pi, instr_\pi \rangle \tag{1}$$

$form_\pi$ symbolically captures all inputs leading to the execution path π, $brdiv_\pi$ captures the total branch divergence along the path π and $instr_\pi$ is the total number of instructions executed along π.

We compute $brdiv_\pi$ per barrier interval. A barrier interval is the code between the start of the kernel and the first barrier instruction or between two consecutive barrier instructions. The end of a kernel serves as an implicit barrier. For each execution path π explored via symbolic execution, the branch divergence per barrier interval b is computed as follows:

$$brdiv_\pi(b) = \frac{divergent\_sets_b}{X-1} \times 100$$

Where $divergent\_sets_b$ is the number of different paths taken by the X threads in the given barrier interval b. Finally, the total branch divergence is computed by summing up the branch divergence over all barrier intervals ($|BI|$ captures the number of barrier intervals):

$$brdiv_\pi = \frac{\sum_{b=1}^{|BI|} brdiv_\pi(b)}{|BI|}$$

Branch divergence and the number of instructions per execution path serve two crucial information for guiding the test generation. In general, the genetic algorithm systematically leverages the information on branch divergence and number of instructions in order to generate test inputs maximizing kernel execution time.

B. Scaling

In this stage, we scale the input atoms generated via symbolic execution to fit the input size of the given GPU kernel.

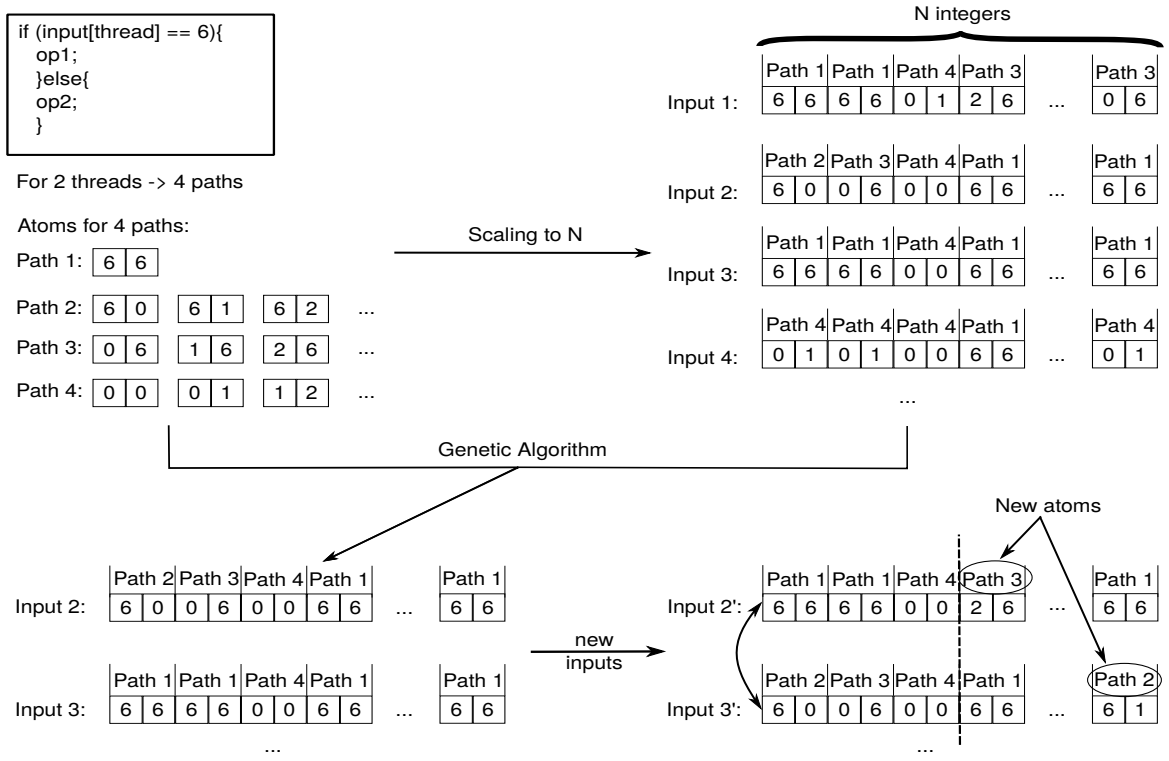


Fig. 3. Examples on scaling to large inputs and creating new inputs during the Genetic Algorithm stage

Assuming that the GPU kernel involves N GPU threads, recall that we run the symbolic execution for $X \leq N$ threads. As a by-product of symbolic execution, we obtain a set of paths (captured by symbolic formulas) in the GPU kernel involving X number of GPU threads. The key intuition of this scaling process is that often the different paths in the given GPU kernel (that involves N threads) can be generated via the combinations of paths obtained from its trimmed version (that involves X threads). For instance, consider our examples in Figure 1. Assume $path_n^i$ captures the i-th path in the kernel involving n threads. The following relationships hold:

- $path_2^1 = path_1^1 \parallel path_1^1$
- $path_2^2 = path_1^2 \parallel path_1^2$
- $path_2^3 = path_1^1 \parallel path_1^2$
- $path_2^4 = path_1^2 \parallel path_1^1$

where $\parallel$ captures an ordered (with respect to thread identities) combination of different execution paths. Ideally N is divisible by X so that no truncation is needed while scaling the input atoms.

It is worthwhile to mention that branch conditionals targeting thread ID (i.e. $(threadID == CONSTANT)$) or positioning of the thread (i.e. $(threadID == inputLength)$) are special cases that cannot be handled via the scaling process. For instance, when scaling from X threads to $2 \cdot X$ threads, thread X will no longer be equal to the input length. Hence, our trimmed version of symbolic execution will not cover all branch legs for such conditional branches involving input lengths or specific thread IDs. Hence, we complement both the symbolic execution and the scaling process via a genetic algorithm. This is to systematically explore the input space for covering worst-case scenarios that were not obtained after the scaling.

C. Genetic Algorithm

Our engineered algorithm involves a mapping between terms used in genetic algorithm [16] (GA) literature and the specific context of our targeted problem. Table I provides an outline of this mapping and we use the terms used in Table I for the rest of the discussion. Our GA process is outlined in Figure 4.

WCET problem	GA term
Path formula (i.e. $form_\pi$)	Allele
Part of kernel input	Gene
Kernel input	Individual/Chromosome
Kernel execution time	Fitness

TABLE I
MAPPING TO GENETIC ALGORITHM TERMS

An individual is mapped to an input for the kernel. If the kernel runs for N threads and the trimmed symbolic execution runs for X threads, then we encode the chromosome as a combination of $\frac{N}{X}$ genes. Figure 3 captures this phenomenon. In particular, instead of N genes, we have $\frac{N}{2}$ genes to construct $Input\ 1$ as follows:

$$Input\ 1 = Path\ 1 \| Path\ 1 \| Path\ 4 \| Path\ 3 \| \ldots \| Path\ 3$$

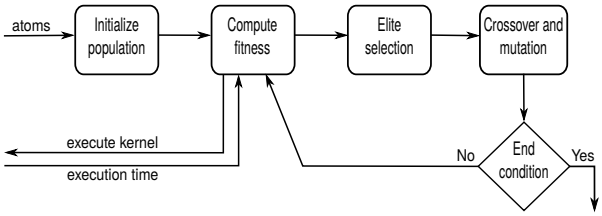


Fig. 4. Stages of the Genetic Algorithm

To maintain diversity in the initial test population, we employ multiple strategies. Firstly, we create individuals by combining a set of randomly selected paths. These paths belong to the trimmed version of the kernel. Secondly, we select individuals by combining paths that manifested maximum number of instructions and branch divergence in the trimmed kernel. Recall that the information on the number of executed instructions and branch divergence was collected during symbolic execution (*cf.* Equation 1). As an example, assume that the path "$Path\ 2$" exhibits the maximum number of instructions and branch divergence. Therefore, we create an individual as follows: $Path\ 2 \| Path\ 2 \| \ldots \| Path\ 2$. The individuals in the initial population, that are not suitable to expose the WCET, are removed in subsequent iterations via the natural selection of the genetic algorithm.

The elite selection stage selects a predefined percentage of individuals from the population based on their fitness (i.e. the kernel execution time). These individuals are kept unaltered to use in the next population. During crossover, we select one of the parents randomly and the other with a bias towards the elite. Subsequently, an 1-point crossover is employed between parents. An example of such an 1-point crossover can be observed in Figure 3 (in the bottom half). A small fraction of individuals are mutated at every iteration of the genetic algorithm. To this end, we implement a low-cost mutation operation. Concretely, we generate two random numbers – the first number to check if we hit the probability to mutate and the other (in case we mutate) to identify the specific gene to mutate.

The iterative process of GA continues until the kernel execution times, manifested by two consecutive generations of GA, do not change substantially. The process is also terminated if the time budget of testing is reached.

D. Why `GDivAn` works?

The reason `GDivAn` works is because of the synergistic combination of symbolic execution (SE) and genetic algorithm (GA). The purpose of our symbolic execution step is not to explore all different aspects in the GPGPU program that may impact the program performance. Due to the complexity of symbolic execution, such a strategy is unlikely to scale. Besides, symbolic executors are classically not designed to explore the performance behaviour of GPGPU programs. As a result, integrating GPU-specific performance features into a symbolic executor would require heavy engineering of state-of-the-art symbolic execution tools. To address such challenges, we propose to use off-the-shelf symbolic executors and explore the structure of GPGPU programs considering branch divergence. This leads to an initial population of test cases for our genetic algorithm. Our genetic algorithm, then, searches the input space and discovers inputs that lead to slower execution times due to other micro-architectural features (e.g. memory coalescing, memory-bank conflicts and cache misses). This makes `GDivAn` a scalable and effective tool to discover the likely WCET of arbitrary GPGPU programs.

V. Evaluation

Experimental setup. We use GKLEE [17] as the symbolic execution tool for `GDivAn`. GKLEE is a symbolic analyzer and test generator tool tailored for CUDA C++ programs. We modify the source code of GKLEE to obtain the relevant information (e.g. branch divergence, number of instructions) for each explored path and drive the genetic algorithm stage within `GDivAn`.

Program name	#Kernels	LOC	#Kernel invocations	#if stmts.	# loops	#threads
`LBM`	1	97	1	13	0	32768
`BFS`	2	17	>1	2	1	1024
		11	>1	1	0	1024
`NSICHNEU`	1	2346	1	252	0	4096

TABLE II
KERNEL PROPERTIES

We have picked three kernels for evaluating `GDivAn`. Specifically, we have chosen kernels involving multiple programs paths to stress test the mechanism implemented within `GDivAn`. Table II captures some salient properties of the chosen subject programs. `NSICHNEU` is a single-threaded CPU program obtained from Mälardalen WCET benchmarks [18]. `NSICHNEU` exhibits complex control flow and implements the simulation of a Petri net. We have modified it to a multi-threaded CUDA program, where each thread of the GPU runs the simulation of a Petri net. `LBM` is a GPU program for computational fluid dynamics using Lattice Boltzmann Models. `BFS` is the GPU implementation of breadth-first search and it is obtained from the Rodinia 3.1 benchmark suite [19].

All the kernels have been evaluated on an NVIDIA Tegra K1 GPU. The kernels were compiled with CUDA nvcc version 6.5. For measuring the execution time on the Tegra K1, the default frequencies have been used, i.e., 72 MHz for the GPU's core clock and 204 MHz for the GPU's memory clock. Each generated test was executed ten times in the GPU and the average of these ten runs are reported in the evaluation.

Evaluating the hypothesis of `GDivAn`. To evaluate the key hypothesis of `GDivAn`, we have implemented a synthetic kernel as shown in Listing 1 to run with 2^{15} threads. This kernel is similar to the example in Figure 1(b) where branch divergence does *not* lead to the WCET of the kernel. In this section, we will refer to this kernel as `Artificial`.

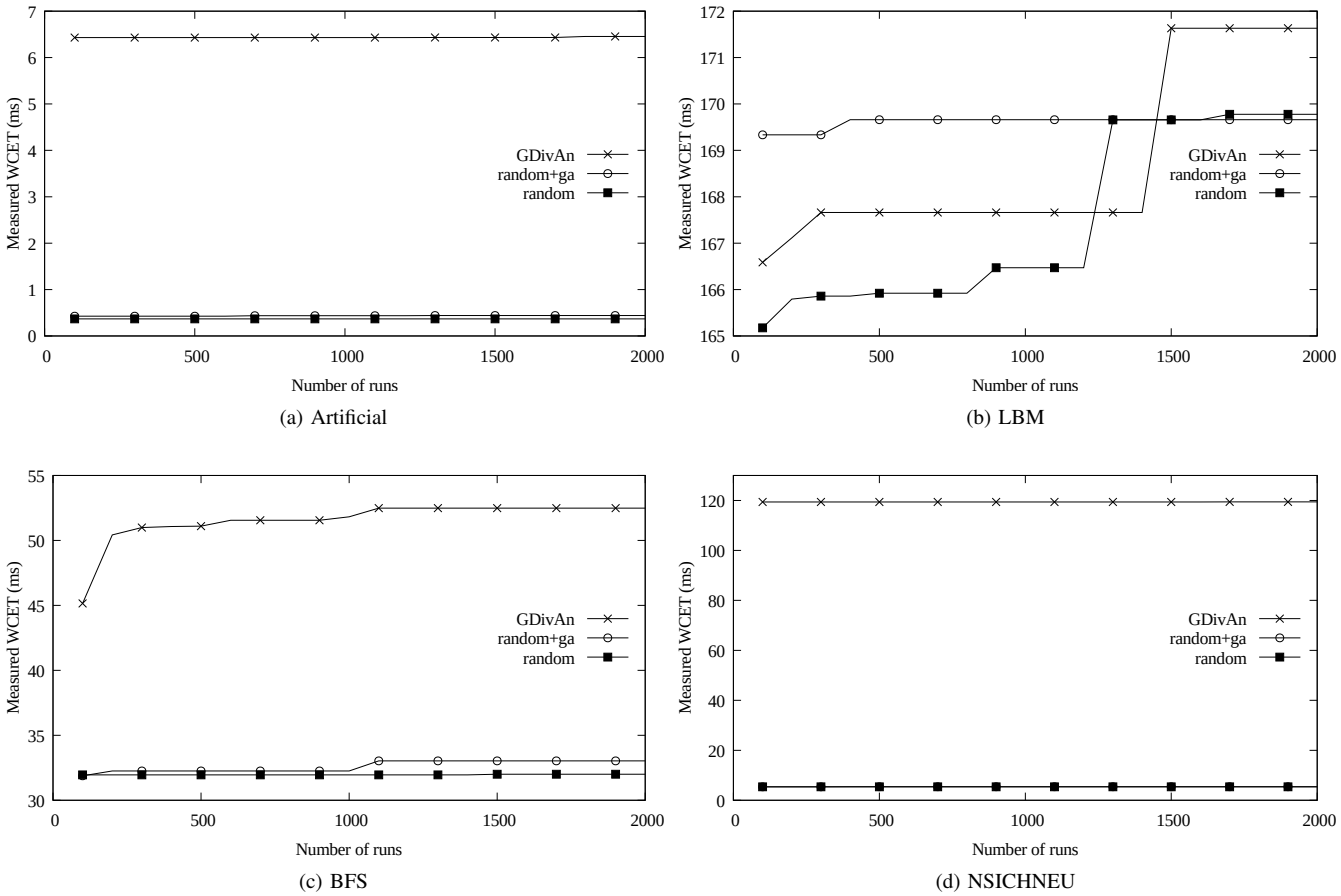


Fig. 5. WCET of kernels via Random, Random+GA, and GDivAn

```
__global__ void kernel(int *values, int *result){

  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (values[tid] == CONSTANT){
    atomicAdd(&result[blockIdx.x], values[tid]);
    ...
    atomicAdd(&result[blockIdx.x], values[tid]);
  }else{
    atomicAdd(&result[tid], 10);
  }

}
```

Listing 1. An example to test the hypothesis of GDivAn

Symbolic execution stage. Table III captures the maximum number of threads GKLEE can run within an hour, the branch coverage obtained and the total time taken to symbolically execute the respective programs. Recall (see Section IV-A) that our goal is to achieve a branch coverage as close as possible to 100%. For Artificial this has been achieved with two threads and the execution time is 1 second. For the other benchmarks, the branch coverage and execution time is indicated with the maximum number of threads that could be analyzed in less than an hour. We have to mention here that the number of threads to be considered is different for each benchmark. LBM, for example, works on square matrices and each element is assigned to one thread. Thus, the next level after four threads would be nine which, however, lead to an analysis time beyond one hour. Similar considerations apply to BFS and NSICHNEU.

Recall that we use the paths explored by GKLEE to create individuals for the genetic algorithm (Section IV-B).

Program	# threads	branch coverage (%)	execution time (secs)
Artificial	2	100	1
LBM	4	80	10
BFS	2	83.33	1
NSICHNEU	2	70.24	341

TABLE III
EVALUATION OF GKLEE RUNS WITH OUR SUBJECT PROGRAMS

Genetic algorithm stage. We select a population size of 100 (i.e. number of inputs in one generation) to run our genetic algorithm. Moreover, we keep the elite percentage 10% (cf. Section IV-C) and the probability to mutate a gene (for a given individual) to be 25%. It is worthwhile to note that we mutate only one gene of an individual, thus keeping the overall mutation rate (across all genes of the individuals) quite low. Finally, while creating the initial population of individuals, we reserve 30% population for individuals with special traits.

	GDivAn		random+ga		random	
Program name	WCET (ms)	WCET reached after (s)	WCET (ms)	WCET reached after (s)	WCET (ms)	WCET reached after (s)
Artificial	6.454	459	0.44	258	0.369	1
LBM	171.631	12028	169.661	1514	169.777	7255
BFS	52.482	1411	33.03	911	32	1128
NSICHNEU	119.412	12669	5.419	13333	5.381	2702

TABLE IV
TESTING TIME. ALL EXPERIMENTS WERE PERFORMED ON AN INTEL I5 MACHINE HAVING 16GB RAM AND RUNNING UBUNTU 16.04

These special individuals were created from paths that exhibited maximum number of instructions and branch divergence during the symbolic execution. The parameters of the GA has been set after a preliminary set of extensive experiments.

Overall evaluation. Figure 5 outlines the overall evaluation of GDivAn. To stress test our approach, we compare GDivAn with both random testing ("random" in Figure 5) and our genetic algorithm without the symbolic execution step ("random+ga" in Figure 5). For "random+ga" approach, we created the initial population of genetic algorithm randomly. For a fair comparison, the number of test runs across all approaches is kept the same. Figure 5 clearly shows that GDivAn approach outperforms the rest in terms of exposing the kernel WCET.

Table IV captures the WCET detected with the three approaches. The quality of WCETs obtained via GDivAn is significantly higher than that produced via "random" and "random+ga". Table IV also highlights the time consumed for each approach until it reached the WCET it was able to produce (after that time no improvements on the measured WCET were observed). The analysis time reported in Table IV is the sum of the GA steps, the time for data allocation and copying on the GPU, and the kernel execution time. The kernel execution time takes from around 10% (for NSICHNEU) to around 33% (for BFS) of the total analysis time, until GDivAn reports the WCET of the respective programs. Note that this kernel execution time already accounts running the respective kernel ten times for each input. For GDivAn, the indicated time also includes the duration of the symbolic execution (see Table III).

VI. DISCUSSION

In this paper, we propose GDivAn, a novel approach to test the worst-case execution time (WCET) of arbitrary GPGPU programs. GDivAn systematically combines the strength of symbolic execution and genetic algorithm to converge towards the WCET. We evaluate GDivAn with several GPU kernels and show its effectiveness compared to both random testing and genetic algorithm in their pure forms. In the future, we would like to investigate the capability of GDivAn to compute the response time of arbitrary GPU-based applications represented as task graphs. We also plan to use GDivAn for driving worst-case oriented optimizations of GPU kernels. To facilitate reproducibility and further research on the subject, our implementation and experimental data are publicly available here: https://bitbucket.org/AdrianHorga/gdivan

REFERENCES

[1] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. C. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *RTAS*, 2017, pp. 353–364.

[2] G. A. Elliott and J. H. Anderson, "Real-world constraints of GPUs in real-time systems," in *RTCSA*, 2011, pp. 48–54.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[4] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *RTSS*, 2007, pp. 49–60.

[5] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified WCET analysis framework for multicore platforms," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4s, pp. 124:1–124:29, 2014.

[6] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *RTSS*, 2013, pp. 33–44.

[7] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "WCET measurement-based and extreme value theory characterisation of CUDA kernels," in *RTNS*, 2014, pp. 279:279–279:288.

[8] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar, "Measurement-based probabilistic timing analysis for graphics processor units," in *ARCS*, 2016, pp. 223–236.

[9] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston *et al.*, "Proartis: Probabilistically analyzable real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, p. 94, 2013.

[10] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, "Open challenges for probabilistic measurement-based worst-case execution time," *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 69–72, 2017.

[11] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart, "On the sustainability of the extreme value theory for WCET estimation," in *OASIcs-OpenAccess Series in Informatics*, vol. 39. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[12] G. Lima, D. Dias, and E. Barros, "Extreme value theory for estimating task execution time bounds: A careful look," in *ECRTS*, 2016, pp. 200–211.

[13] A. Betts and A. Donaldson, "Estimating the WCET of GPU-accelerated applications using hybrid analysis," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 193–202.

[14] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos, "Symbolic search-based testing," in *ASE*, 2011, pp. 53–62.

[15] "CUDA toolkit documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[16] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[17] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: concolic verification and test generation for GPUs," in *PPOPP*, 2012, pp. 215–224.

[18] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *WCET*, 2010, pp. 137–147.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.