Automated Software Testing of Memory Performance in Embedded GPUs

Sudipta Chattopadhyay Petru Eles Zebo Peng
Linköping University
{sudipta.chattopadhyay,petru.eles,zebo.peng}@liu.se

Abstract

Embedded and real-time software is often constrained by several temporal requirements. Therefore, it is important to design embedded software that meets the required performance goal. The inception of embedded graphics processing units (GPUs) brings fresh hope in developing high-performance embedded software which were previously not suitable for embedded platforms. Whereas GPUs use massive parallelism to obtain high throughput, the overall performance of an application running on embedded GPUs is often limited by memory performance. Therefore, a crucial problem lies in automatically detecting the inefficiency of such software developed for embedded GPUs. In this paper, we propose GUPT, a novel test generation framework that systematically explores and detects poor memory performance of applications running on embedded GPUs. In particular, we systematically combine static analysis with dynamic test generation to expose likely execution scenarios with poor memory performance. Each test case in our generated test suite reports a potential memory-performance issue, along with the detailed information to reproduce the same. We have implemented our test generation framework using GPGPU-Sim, a cycle-accurate simulator and the LLVM compiler infrastructure. We have evaluated our framework for several open-source programs. Our experiments suggest the efficacy of our framework by exposing numerous memory-performance issues in a reasonable time. We also show the usage of our framework in improving the performance of programs for embedded GPUs.

1. Introduction

Embedded and real-time software is often required to satisfy several timing constraints. Therefore, validating the performance of embedded software is a critically important problem. Performance validation methodologies aim to *automatically* detect corner scenarios that capture poor performance behaviour of embedded software. In the absence of such validation techniques, an embedded software may suffer from serious performance failure at runtime. As a result, performance validation methodologies are required early during the design process of the software. In this way, the reported performance issues can be fixed to avoid runtime failure.

Graphics processing units (GPUs) have gained popularity as a flexible and efficient solution for parallel processing. During the last decade, researchers have shown significant performance gains that could be achieved using mainstream GPUs. Consequently, many embedded devices (*e.g.* smartphones) are now equipped with programmable GPUs. A few examples of such embedded GPUs include graphics processing cores designed by Vivante, ARM and AMD. As primarily targeted for power-constrained devices, embedded GPUs often have very restricted features compared to mainstream GPUs. Such restrictions include small caches, limited or no support of low-latency shared memory, among others. Therefore, careful design choices are required to develop high-performance embedded software on embedded GPUs [23].

One appealing feature of general-purpose GPU (GPGPU) programming is the availability of powerful programming abstractions. For instance, to perform a certain computation by GPU, a programmer writes the code for only one *GPU thread* and specifies the number of threads to execute. On one hand, such programming abstractions provide developers tremendous flexibility in writing parallel modules. On the other hand, due to such layer of programming abstractions, the execution pattern of a GPGPU application remains *completely hidden* from the developer. As a result, it becomes potentially impossible for a developer to detect and understand the hidden performance problems in an embedded GPGPU application. As pointed out by existing literature, memory subsystems may significantly limit the performance gain from parallelism offered by GPUs [17, 18]. Accessing the global-memory (DRAM) in embedded GPUs is several magnitudes slower than accessing on-chip memories (*e.g.* caches). Although such slow memory accesses are partially overlapped by computations, an application launched in the GPU may suffer from performance problems due to the congestion in the memory controller or caches [18]. In embedded GPUs, such performance problems might be critical due to limited on-chip memories. Moreover, detecting such problems is challenging due to the presence of a huge number of possible execution scenarios (*e.g.* different inputs and thread schedules) and it is potentially infeasible to run an application for all such scenarios. Therefore, it is highly desirable that such performance problems are systematically detected and highlighted to the developer via *automated software testing*.

In this paper, we propose GUPT, an automated test generation framework to systematically detect performance problems in embedded GPGPU applications. In particular, our framework systematically explores execution scenarios which may create substantial interferences from different threads in the memory subsystem (*e.g.* memory controller and on-chip caches) and prolong the overall execution time of the respective application. We generate a test suite where each test case highlights a specific execution scenario (*i.e.* inputs and thread schedules) that leads to such inefficiencies in the GPGPU program. Systematically detecting such performance issues is challenging for several reasons. First of all, application code is, in general, not annotated with any extra-functional proper-

[Copyright notice will appear here once 'preprint' option is removed.]

ties (*e.g.* time). Besides, a GPGPU program has a potentially large number of execution scenarios. As a result, any naive test generation strategy is either *infeasible* in practice (*e.g.* exhaustive testing) or it may lead to an extremely low coverage of the potential performance problems. In our framework, we solve these challenges by first producing a summary of all GPU threads via static analysis. Subsequently, this summary is used to guide the test generation to produce memory-performance stressing test cases. Although targeted for embedded GPGPU applications, we believe that such a test generation strategy is also useful for other embedded software, such as applications running on embedded multi-core processors.

To guide a test generation that stresses memory-performance, it is critically important to identify the causes of memory performance issues. In this work, we primarily focus on performance issues that arise due to the interferences created by different threads. We broadly classify such interferences into *direct interferences* and *indirect interferences*. A *direct interference* can be described as a scenario where a substantial number of threads access the DRAM, creating *memory congestion* and affecting the overall execution time. An *indirect interference* is caused when GPU threads replace each others content in on-chip caches, resulting in substantial traffic to the DRAM. To systematically explore such scenarios, it is crucial to compute the potential locations of DRAM accesses and on-chip cache states of different GPU threads. We accomplish this by statically analyzing a GPGPU program. The static analyzer produces a summary of different GPU threads. Such a summary does not take into account the possible interference among threads and it is computed *only once*. Typical information in this summary includes potential locations of DRAM accesses and cache states in GPU threads. Once this summary is computed, we invoke a guided test generation. The general strategy behind this guidance is to generate appropriate interferences in the on-chip caches and DRAM which may affect the overall execution time.

Our test generation revolves around systematically exploring execution states using the summary generated by static analyzer. We first execute the GPGPU program under test with a random input and we compute a symbolic state for each GPU thread. Such a symbolic state captures the set of all inputs that lead to the respective execution scenario of the thread. In GPUs, threads are often scheduled in batches (*e.g. warps* in CUDA terminologies). In such cases, we merge the symbolic states from different threads of the same group (*e.g. warp* in CUDA) and produce a symbolic state for each group. To generate a new execution scenario that is likely to lead to a memory-performance issue, we employ several strategies. We manipulate the symbolic state in such a fashion that the resulting input is likely to lead to substantial DRAM traffic. For instance, we generate a new input that follows a control dependence edge (in the control dependence graph of the GPGPU program) which leads to potentially high DRAM accesses. Besides, while scheduling threads (or groups of threads), we select threads (or groups of threads) that may issue potentially high DRAM requests or create substantial on-chip cache conflicts. We monitor each execution to record the DRAM congestion or on-chip cache conflicts. We continue exploring different execution scenarios in a similar fashion, as long as the time budget dedicated to testing permits or all execution states with potential DRAM accesses are visited. As we target memory-performance, the quality of our generated test suite is provided via the coverage of potential DRAM access locations in each thread.

Contribution In summary, we have proposed and developed a test generation framework to expose memory-performance bottlenecks in embedded GPGPU applications. Due to the limited availability of on-chip memories in embedded GPUs, such performance bottlenecks should be addressed in order to develop high performance embedded software. Our proposed framework uses the power of both static analysis and dynamic test generation to expose as many performance issues as possible for a given time budget. Each test case in our generated test report highlights a memory performance issue and the respective execution scenario (i.e. raw inputs and thread schedules) that leads to the same. The primary use of our framework is to help developer write efficient embedded applications by reporting the existing performance issues. Besides, our generated test-suite can be run on a specific embedded GPU and the replicated problems can be investigated by existing GPU-based profilers. We have implemented our entire framework on top of GPGPU-Sim [8], LLVM compiler infrastructure [4] and GKLEE symbolic virtual machine [22]. Specifically, GPGPU-Sim is used to implement the static analyzer and monitor the execution of a GPGPU program. The LLVM compiler and GKLEE are used to compute the symbolic state for a specific execution scenario. Our evaluation with several CUDA SDK kernels [2] reveals that our framework uncovers memory performance issues early during the test generation. In particular, some of our evaluations reveal that algorithmically more efficient (low computational complexity) GPU implementation may lead to poor performance compared to a naive GPU implementation of the same functionality. This happens due to the inherent memory interferences at the micro-architecture level. Finally, we show the usage of our framework to improve the performance of some GPGPU programs.

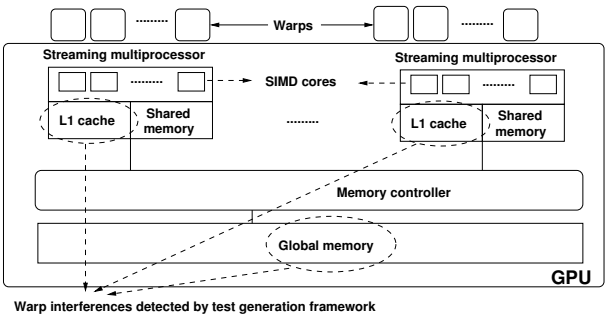

Figure 1. An abstract view of the GPU targeted by our framework

2. System Architecture

Figure 1 shows the relevant portions of a GPU architecture targeted by our test generation framework. The GPU contains a number of streaming multiprocessors (SM). Each SM contains several cores to execute batches of GPU threads. Besides, each SM contains an on-chip L1 cache and it may also contain an on-chip shared-memory. In this work, we do not consider the presence of an L2 cache. Therefore, the global-memory (DRAM) needs to be accessed for an L1 cache miss. The architecture shown in Figure 1 captures an abstract view of typical embedded GPUs. For instance, the embedded GPU designed by Vivante [6] has a similar architecture, except the presence of an on-chip shared memory.

The input to our test generation framework is the implementation of a GPU kernel using CUDA[1] [3]. Each SM in the GPU fetches and schedules a batch of GPU threads (called *warp* in CUDA terminologies). At any instant, one or more warps might be active on the same SM. Finally, we assume that the *warp scheduler* is invoked only if a running warp is *blocked* (*e.g.* due to DRAM transaction or synchronization barrier). The warp scheduler selects a warp from a pool of all warps ready for execution and issues the warp into an SM. It is worthwhile to note that the exact selection strategy of such a warp scheduler is typically *unknown* and it may

[1] Our framework is equally applicable for OpenCL kernels. However, our implementation currently supports only CUDA kernels.

also vary depending on the specific execution platform. Moreover, even for a specific execution platform, the warp scheduler may expose non-deterministic behaviour, as discussed in [19]. Therefore, without loss of generality, we shall only consider the non-deterministic behaviour in warp selection strategy.

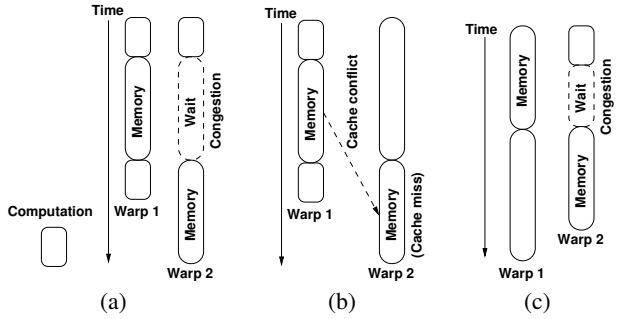

Figure 2. (a) DRAM congestion, (b) inter-warp cache conflict, (c) congestion overhead does not affect the overall execution time

In this work, we specifically detect memory-performance issues due to inter-warp interferences. This might appear due to *(i)* DRAM congestion, or *(ii)* additional cache misses due to inter-warp conflicts. As pointed out by recent literature [18], this is a critical issue which often limits the potential performance gain from GPU. Therefore, during the test generation process, we have overhead in the following two scenarios:

- A warp waits for a global-memory transaction due to other warps accessing the global-memory. We call this scenario a *direct interference*.
- A warp faces additional L1 cache misses due to the on-chip cache conflicts generated by other warps. We call this scenario an *indirect interference*.

Figure 2(a) and Figure 2(b) show examples of a *direct interference* and an *indirect interference*, respectively, assuming that DRAM can service only one memory transaction at a time. Figure 2(c) captures a scenario where the overall execution time is not affected even in the presence of inter-warp interferences. Our framework does not report such scenarios, as the overall execution time is unaffected by the memory interference.

3. Overview

In this section, we shall illustrate our test generation framework via an example. For the sake of simplicity in this example, we shall generically describe a thread as the scheduling unit in a GPU. However, our methodology is equally applicable where the scheduling unit is a group of threads and our current implementation considers a *warp* (in CUDA terminologies) as the scheduling unit.

Figures 3(a)-(c) show the control flow graphs (CFGs) of three different GPU threads $T1$, $T2$ and $T3$ from a GPU kernel. It is worthwhile to note that different GPU threads in a GPU kernel have the same CFG, which captures the static control flow of the GPGPU code. However, these threads may access different data. Figures 3(a)-(c) show three such GPU threads which access different memory blocks via their identities $T1, T2$ and $T3$. These thread identifiers are captured via tid. For the sake of illustration, we shall assume a direct-mapped cache, a single-bank DRAM and the following cache-mapping pattern: $\{m1, m4\} \mapsto \mathcal{S}_1$, $m2 \mapsto \mathcal{S}_2$, $\{m3, m5\} \mapsto \mathcal{S}_3$, $\{m6, m8, m9\} \mapsto \mathcal{S}_4$, $m7 \mapsto \mathcal{S}_5$. Therefore, memory blocks $m1$ and $m4$ (and similarly, memory blocks $m3, m5$ and $m6, m8, m9$) can evict each other from the cache.

To generate memory-performance stressing test cases, we first perform a static analysis of the GPGPU code and extract a *summary* of each thread. This static analysis is light-weight in the sense that it analyzes each thread *in isolation* and therefore, it does not take into account any interference between threads. The interference between threads is later considered during the test generation process. The primary purpose of the static analyzer is to extract relevant information that can guide the test generation process. Specifically, the summary computed by the static analyzer contains three different information: *(i)* a set of locations in each thread where DRAM might be accessed, *(ii)* possible reuse of cache contents at different program locations, and *(iii)* cache conflicts generated by each thread. This summary is utilized to generate execution scenarios that may potentially exhibit direct and indirect memory interferences (*cf.* Section 2). According to [25], we classify memory references as AH (always in the cache when accessed), AM (never in the cache when accessed) or PS (never evicted from the cache). If a memory reference cannot be classified as AH, AM or PS, it is categorized as *non-classified* (NC). Figures 3(a)-(b) show this categorization beside each memory reference. It is important to note that AM and NC categorized memory references are the potential locations for frequent DRAM requests. The reuse of cache contents and cache conflicts can be computed via backward data flow analysis. For instance, consider the exit point of basic block $B5$. Thread $T1$ can reuse only $\{m1, m2, m3\}$ from the cache, as $m4$ is *always* evicted by $m1$ before being reused from the cache (*cf.* Figure 3(a)). However, for thread $T3$, all the memory blocks $\{m1, m2, m5\}$ might be reused (*cf.* Figure 3(b)). This is due to the absence of any intra-thread cache conflicts in $T3$.

We start our test generation process by executing the GPU kernel for a random input. Let us assume that we start with a random input $\langle a[T1 \ldots T3] = 0, b[T1 \ldots T3] = 0\rangle$. Once the kernel finishes execution, we extract the symbolic state for each thread to capture the respective execution scenario. For instance, the following three conditions capture the initial execution scenarios of $T1$, $T2$ and $T3$: *(i)* $\Phi_{T1} : a[T1] \leq 0 \wedge b[T1] \leq 2$, *(ii)* $\Phi_{T2} : a[T2] \leq 0 \wedge b[T2] \leq 2$ and *(iii)* $\Phi_{T3} : a[T3] \leq 0 \wedge b[T3] \leq 2$.

Let us now assume that we want to stress memory-performance via indirect interferences (*cf.* Figure 2(b)). For each thread, we first choose an appropriate symbolic state that is likely to generate on-chip cache conflicts. To accomplish this, we manipulate the symbolic states from previous execution scenarios. For instance, consider thread $T1$. We observe that memory block $m3$ creates cache conflicts to thread $T3$ and access to $m3$ is control dependent on the branch condition $b[T1] > 2$ (*cf.* Figure 3(d)). Therefore, we negate the branch condition $b[T1] \leq 2$ from the previous execution to generate the symbolic state $\Phi'_{T1} \equiv a[T1] \leq 0 \wedge \neg(b[T1] \leq 2)$ for thread $T1$. In a similar fashion, in $T3$, since memory block $m5$ creates cache conflicts to $m3$ and access to $m5$ is control dependent on the branch condition $b[T3] \leq 2$ (*cf.* Figure 3(d)), we use the symbolic state $\Phi_{T3} \equiv a[T3] \leq 0 \wedge b[T3] \leq 2$ from the initial execution. In general, we use the control dependence graph (CDG) to find a symbolic state that leads to the maximum inter-thread conflicts. For instance in $T3$ (*cf.* Figure 3(d)), only the control dependence edge $\langle B4(m2), B6(m5)\rangle$ leads to inter-thread conflicts, due to the memory reference $m5$. As a result, for $T3$, we choose a symbolic state that satisfies the condition to execute $\langle B4(m2), B6(m5)\rangle$ (*i.e.* $b[tid] \leq 2$). Thread $T2$ does not create any inter-thread conflicts. Therefore, we choose a symbolic state that may potentially generate more memory traffic. Note that $m8$ and $m9$ are classified as AM. Since access to $m8$ and $m9$ are control dependent on the conditional $a[T2] \leq 0$ and $b[T2] \leq 2$ (*cf.* Figure 3(d)), respectively, we choose the symbolic state Φ_{T2} for thread $T2$. Finally, to execute the kernel, we generate a concrete input satisfying the formula $\Phi'_{T1} \wedge \Phi_{T2} \wedge \Phi_{T3}$.

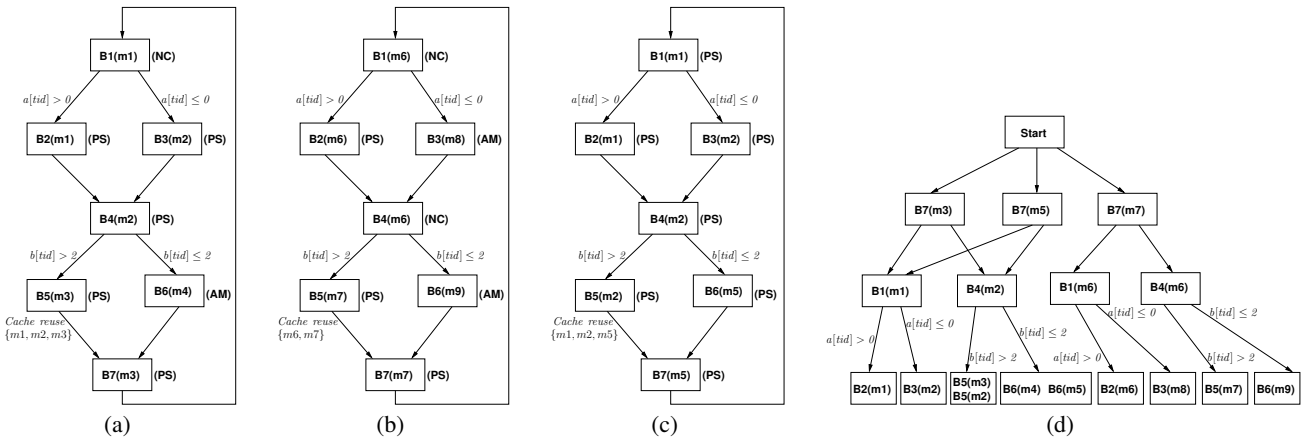

Figure 3. Control flow graphs (CFGs) of thread (a) $T1$, (b) $T2$ and (c) $T3$. (d) Control dependence graph (CDG) of the GPU kernel. Variable tid captures thread identity. The annotation $Bx(mx)$ captures the basic block identity as Bx and the memory block mx accessed therein.

In the preceding paragraph, we have outlined the technique to generate new test inputs for stressing the memory-performance. Apart from generating new test inputs, it is also important to systematically schedule threads, so that it is likely to lead to memory-performance problems. For instance, let us consider an execution scenario for a test input $\theta \in \Phi'_{T1} \wedge \Phi_{T2} \wedge \Phi_{T3}$ and the time where thread $T1$ is blocked to fetch $m3$ from DRAM. At this point, we schedule a thread which creates maximum cache conflicts to the *reused cache content*. Note that $T1$ may reuse $\{m1, m2, m3\}$ from the cache (*cf.* Figure 3(a)) after it becomes ready for execution. Thread $T2$ does not create any conflict to the reused cache content $\{m1, m2, m3\}$, whereas accessing memory block $m5$ in $T3$ creates conflict to $m3$. As a result, if both $T2$ and $T3$ are ready for execution, we schedule $T3$ when $T1$ is blocked to fetch $m3$ from DRAM. In a similar fashion, we schedule $T1$ when $T3$ is blocked to fetch $m5$ from DRAM. Continuing in this way, for a test input $\theta \in \Phi'_{T1} \wedge \Phi_{T2} \wedge \Phi_{T3}$, threads $T1$ and $T3$ keep replacing cache contents and generate potentially high DRAM traffic. We continue to generate new test inputs and thread schedules in this fashion as long as the time budget dedicated to testing permits.

The overall intuition behind the generation of direct memory interferences is similar to the technique outlined in the preceding paragraphs. In this case, the cache access categorization (*i.e.* AH-AM-PS-NC categorization) guides the test generation process. Specifically, the symbolic states and thread schedules are generated in such a fashion that it leads to a substantial number of NC or AM classified memory references. For instance, we select the symbolic states $a[tid] \leq 0 \wedge b[tid] \leq 2$ for both $T1$ and $T2$. This is because, memory references $m4$, $m8$ and $m9$ are categorized as AM and access to $m4, m8, m9$ are control dependent on the branch conditional $a[tid] \leq 0$ and $b[tid] \leq 2$ (*cf.* Figure 3(d)). Besides, if both $T1$ (or $T2$) and $T3$ are simultaneously ready for execution, we schedule $T1$ (or $T2$) to generate more DRAM traffic.

It is worthwhile to note that even for this simple example, it is non-trivial to systematically generate test inputs that expose memory-performance problems. For instance, consider any input that satisfies the symbolic condition $\bigwedge_{t \in \{T1,T2,T3\}} a[t] > 0 \wedge b[t] > 2$. Such inputs do not generate any memory-performance problems. This is due to the absence of any inter-thread cache conflicts and DRAM congestion. As a result, any ad-hoc test generation strategy may lead to a poor coverage of the existing performance problems. In a similar fashion, consider any test input $\theta \in \bigwedge_{t \in \{T1,T2\}} (a[t] \leq 0 \wedge b[t] > 2) \wedge a[T3] \leq 0 \wedge b[T3] \leq 2$. If we interleave the execution of $T1$ and $T2$ (instead of $T1$ and $T3$), the resulting schedule may not generate substantial cache conflicts to uncover a potential memory-performance issue. This is because $T2$ does not generate any cache conflict to $T1$. In our proposed framework, the summary computed by the static analyzer systematically guides the test generation process and computes appropriate test inputs, along with pathological thread schedules.

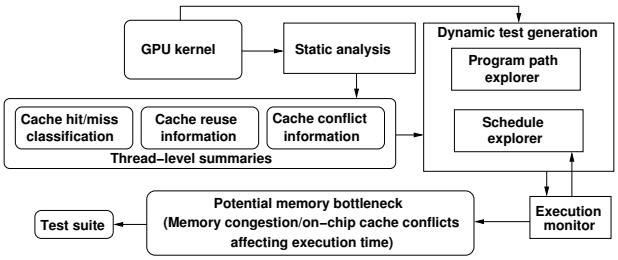

Figure 4. Overall test generation framework

Figure 4 captures our overall framework. Broadly, our framework contains the following components.

- *Static analysis* is used to compute a summary of each GPU thread. Specifically, the static analyzer computes the cache hit/miss classification of memory references, the information about cache reuse and the information about cache conflict. This will be discussed in Section 4.1.
- Test generation component (as shown by "Dynamic test generation" in Figure 4) uses thread-level summaries computed by the static analyzer and identifies appropriate program paths (*Program path explorer*) and thread schedules (*Schedule explorer*) to expose potential memory-performance bottlenecks. This will be discussed in Section 4.2.2.
- *Execution monitor* is used within a cycle-accurate simulator to detect potential memory-performance bottlenecks. This will be discussed in Section 4.3.

The test generation process continues in an iterative fashion until the time budget for testing expires or all execution scenarios with possible DRAM accesses have been explored.

4. Detailed Methodology

In this Section, we shall provide a more detailed description of our framework. For the sake of simplicity in the following discussion,

we shall generically use the term *memory block* for both instruction memory blocks and data memory blocks.

4.1 Static analysis

In this section, we describe the static analyzer in our framework. It works on the CFG of a GPU kernel. This CFG is extracted from the kernel binary (in our implementation, we use CUDA binaries) and investigates different threads *in isolation*. However, since all the threads share the same code, the static analysis can be performed simultaneously for all the threads. Nevertheless, we maintain thread-level information to distinguish the data and control flow in different threads. Since all the threads are analyzed in an exactly same fashion, in the following, we describe the static analyzer only for a single thread.

4.1.1 Cache hit/miss classification

We use abstract interpretation (AI) based cache analyses [12, 25] to classify each memory reference as *all-hit* (AH), *all-miss* (AM), *persistence* (PS) or unclassified (NC). A memory reference is classified as AH (AM), if the respective memory block is always (never) in the cache when accessed. A memory reference is categorized as PS, if the accessed memory block cannot be evicted from the cache. A PS-categorized memory reference can further be decomposed into NC (for its first occurrence) and AH (for all next occurrences). We already perform this decomposition in our framework by virtually unrolling each loop only once. Therefore, in the following discussion, we shall only talk about AH, AM and NC categorized memory references. Cache hit/miss classifications are used primarily to locate a potential global-memory access. Note that AM or NC categorized memory references exhibit potential global-memory accesses by a thread. For further details on cache analyses, readers are referred to [12, 25].

4.1.2 Cache reuse information

We compute the reuse information of caches to create appropriate inter-thread cache conflicts during the test generation. The reuse statistics of caches can be computed via a *backward flow analysis* on the CFG. During cache hit/miss classification, we compute abstract cache states at each location. Such an abstract cache state captures the set of memory blocks which *might be cached* (defined as *may-cache-state* in [25]). We use abstract interpretation (AI) to compute the information about cache reuse. The abstract domain $\mathbb{D}$ is all possible subsets of the set of accessed memory blocks $\mathbb{M}$.

The transfer function τ_b of this AI-based analysis traverses each instruction through a backward control flow and updates the abstract state $\mathcal{D} \in \mathbb{D}$. Let us assume that $\mathcal{C}_p \in \mathbb{D}$ denotes the set of memory blocks which might be cached immediately before the program location p. This was computed during the cache hit/miss classification. We can now formally define τ_b as follows:

$$\tau_b : 2^{\mathbb{M}} \times \mathbb{P} \rightarrow 2^{\mathbb{M}}$$

$$\tau_b(\mathcal{D}, p) = (\mathcal{D} \cup M_{gen}) \setminus (M_p \setminus M_{gen}) \tag{1}$$

where $\mathbb{P}$ denotes the set of all program points, M_p captures the set of memory blocks accessed (by the thread under analysis) at $p \in \mathbb{P}$ and $M_{gen} = \{m \mid m \in M_p \wedge m \in \mathcal{C}_p\}$ (*i.e.* the set of possibly reused memory blocks from the cache at p).

The abstract join operation is employed at a control flow merge point. Specifically, the join operation $\mathcal{J}_b$ is used to compute the abstract state at the exit of a basic block by combining all the abstract states at the entry of its successors. For this analysis, $\mathcal{J}_b$ is primarily a set union operation that merges all the abstract states. We aim to generate cache conflicts that might evict the set of reused memory blocks during the test generation.

We start our analysis with an empty set. Since CFGs usually contain loops, a fixed-point computation is applied to stabilize the abstract state at each program location. For instance, in Figure 3(a), the analysis reaches a fixed-point on $\{m1, m2, m3\}$ at the exit of basic block $B5$. This is because, memory block $m4$ is evicted along all paths reaching the basic block $B5$.

4.1.3 Cache conflict information

This component of the static analyzer is used to estimate the number of cache conflicts generated by a thread between two consecutive blocking points. A thread might be blocked for several reasons, such as due to accessing DRAM or waiting at a synchronization barrier. From a given program location, we compute an estimate on the set of memory blocks that could be accessed by a thread till it goes into the waiting state. This information can also be computed via an AI-based backward flow analysis. The abstract domain of the analysis captures all possible subsets of the set of accessed memory blocks $\mathbb{M}$. The transfer function τ'_b collects the set of accessed memory blocks in set $\mathcal{M}$ via a backward control flow and τ'_b can formally be defined as follows.

$$\tau'_b : 2^{\mathbb{M}} \times \mathbb{P} \rightarrow 2^{\mathbb{M}}$$

$$\tau'_b(\mathcal{M}, p) = \begin{cases} \mathcal{M} \cup M_p, & \text{if } p \notin \mathit{Waiting}; \\ \phi, & \text{otherwise.} \end{cases} \tag{2}$$

where $\mathbb{P}$ denotes the set of all program points, M_p captures the set of memory blocks accessed by the thread at program point p and $\mathit{Waiting} \in \mathbb{P}$ captures the set of potential locations in the thread where it might go into the waiting state. In our analysis, we consider such waiting locations to be either global-memory references (*i.e.* AM or NC classified memory accesses) or synchronization barriers. The join operation simply performs a set union at control flow merge points. We start the analysis with the initial state ϕ and obtain a fixed-point solution at each program location by employing the transfer (*i.e.* τ'_b) and join operation. For instance, in Figure 3(a), the analysis computes a fixed point on $\{m3\}$ at the exit of basic block $B5$. This is due to the reason that $\{m3\}$ is accessed at $B7$ and all paths starting from $B7$ has to access a potential global-memory reference (*i.e.* accessing NC-classified $m1$ at $B1$) next.

4.2 Test generation

Algorithm 1 shows an outline of our test generation and test execution process (*cf.* components "Dynamic test generation" and "Execution monitor" in Figure 4). The basic idea behind this process is as follows. We first generate a random input and execute the GPU kernel under test. Once the kernel finishes execution, we construct a symbolic state ϕ_t for each thread t to capture its execution. In particular, such a symbolic state ϕ_t captures the set of all inputs for which thread t will follow the respective execution path (commonly known in literature as *path condition* [15]). Note that the unit of scheduling in CUDA is a *warp*. Therefore, we compute a warp-level symbolic state Ψ_w by merging the symbolic states of different threads that are contained in the respective warp (line 16 in Algorithm 1). This is accomplished via `Merge` procedure (refer to Section 4.2.1 for details). We manipulate the warp-level symbolic states from previous executions to produce unexplored symbolic states in a list $unex[w]$. More importantly, we prioritize the unexplored symbolic states via procedure `InsertWithPriority` (line 24 in Algorithm 1) using the information computed during static analysis (refer to Section 4.2.2 for details). Lines 9-27 in Algorithm 1 show the methodology to generate unexplored symbolic states and subsequently, prioritizing them after a specific execution of the GPU kernel. To execute the kernel with a new test input, we use the prioritized list of unexplored symbolic states and generate a concrete input from a symbolic state which is more likely to expose memory-performance problem (lines 31-41 in Algorithm 1). Moreover, when the kernel executes with this test input, warps are

Algorithm 1 Performance-stressing test generation for a GPU-kernel

1: **Input:**
2: $\mathcal{P}$: The GPU-kernel under test
3: $Chmc$: Cache hit/miss classification (*cf.* Section 4.1.1)
4: $Reuse$: Cache reuse information (*cf.* Section 4.1.2)
5: $Conflict$: Cache conflict information (*cf.* Section 4.1.3)
6: **Output:**
7: $\mathcal{T}$: A set of test cases. Each test case is of the form $\langle \Omega, \mathcal{SH}, \xi \rangle$, where Ω is the symbolic input-condition, $\mathcal{SH}$ captures the warp-schedule and ξ is the set of inter-warp interferences.
8: Execute the kernel $\mathcal{P}$ on a random input τ
9: Let symbolic state ϕ_t is the path condition ([15]) for thread t
10: /* Compute warp-level symbolic state */
11: Let $\mathcal{W}$ is the set of all warps
12: **for** each warp $w \in \mathcal{W}$ **do**
13: $unex[w] = all[w] = empty$
14: Let $\{t_1, t_2, \ldots, t_n\}$ is the set of threads in w
15: /* Merge thread-level symbolic states */
16: Ψ_w = `Merge`$(\phi_{t_1}, \phi_{t_2}, \ldots, \phi_{t_n})$
17: $unex[w] \bigcup = \{\Psi_w\}$; $all[w] \bigcup = \{\Psi_w\}$
18: Let $\Psi_w = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$
19: **for** $i \in [1, k]$ **do**
20: $\Psi^i_w = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$
21: /* prioritize symbolic states for effective testing */
22: **if** Ψ^i_w is satisfiable *and* $\Psi^i_w \notin all[w]$ **then**
23: $all[w] \bigcup = \{\Psi^i_w\}$
24: `InsertWithPriority`$(unex[w], \Psi^i_w)$
25: **end if**
26: **end for**
27: **end for**
28: /* iterative test generation */
29: **repeat**
30: /* Construct feasible symbolic state for testing */
31: $\Omega = true$
32: **for** each warp $w \in \mathcal{W}$ s.t. $unex[w] \neq$ empty **do**
33: Let $unex[w] := \{\Psi^1_w, \Psi^2_w, \ldots, \Psi^{n-1}_w, \Psi^n_w\}$
34: Let Ψ^i_w has priority over Ψ^{i+1}_w for all $i \in [1, n-1]$
35: Find $\Psi^k_w \in unex[w]$ *s.t.* $\Omega \wedge \Psi^k_w$ is satisfiable and
36: for all $j \in [1, k)$, $\Omega \wedge \Psi^j_w$ is unsatisfiable
37: **if** (Ψ^k_w exists) **then**
38: $\Omega = \Omega \wedge \Psi^k_w$; $unex[w] = unex[w] \setminus \{\Psi^k_w\}$
39: **end if**
40: **end for**
41: Let τ be a concrete input satisfying Ω
42: /* Execute P by pathologically scheduling warps
43: and update unexplored symbolic states */
44: `ExecuteAndMonitor`$(P, \tau, \Omega, \mathcal{T})$
45: **until** for all $w \in \mathcal{W}$, $unex[w]$ = empty or *timeout*
46: Report $\mathcal{T}$ for further investigation

scheduled *on-the-fly* in a fashion which is likely to lead to substantial DRAM congestion or on-chip cache conflicts (refer to Section 4.2.2 for details). Each such execution of the GPU kernel is also monitored to record DRAM congestion overhead and on-chip cache conflicts (*cf.* line 44 in Algorithm 1). This is accomplished via procedure `ExecuteAndMonitor` (refer to Section 4.3 for details). The recorded report is produced for the developer for further investigation. Once the kernel finishes execution, we construct unexplored symbolic states in a similar fashion by manipulating the symbolic state of the last execution. The process of test generation and test execution of a GPU kernel continues as long as the time budget for testing permits or all unexplored symbolic states are visited. In the following, we shall describe some critical components of our framework.

Algorithm 2 Computing warp-level symbolic state

1: **procedure** MERGE$(\phi_1, \phi_2, \ldots, \phi_n)$
2: Let $\phi_i = \psi_{i,1} \wedge \psi_{i,2} \wedge \ldots \wedge \psi_{i,\rho(i)}$
3: Let $\mathcal{B}$ is the set of all control branches
4: $k = 1$
5: **repeat**
6: $\forall b \in \mathcal{B}.\ \alpha[b, k] = true$
7: **for** $j \leftarrow 1, n$ **do**
8: **if** $k \leq \rho(j)$ **then**
9: Let b_j is the control branch respective to $\psi_{j,k}$
10: $\alpha[b_j, k] \bigwedge$=$\psi_{j,k}$
11: **end if**
12: **end for**
13: $k = k + 1$
14: **until** $\forall x \in [1, n].\ k > \rho(x)$
15: /* set warp-level symbolic state */
16: $\Psi_w = \bigwedge_{i,j} \alpha[i, j]$
17: **end procedure**

4.2.1 Merging symbolic states

In Algorithm 1, procedure `Merge` is used (*cf.* line 16 in Algorithm 1) to compute the symbolic state of a warp from the symbolic states of its constituent threads. By computing a symbolic state for the entire warp, we can substantially reduce the number of symbolic states to explore. Algorithm 2 outlines the procedure `Merge`. The general intuition is to align the control branches in the traces of different threads. If a branch can be aligned for a set of threads, the respective branch conditions are merged and treated as a single condition during the generation of test inputs. For instance, consider the set of threads in Figures 3(a)-(c) constitute a warp. If the threads execute for $a[T1 \ldots T3] = b[T1 \ldots T3] = 0$, we compute a warp-level symbolic state $\Psi \equiv \Psi_1 \wedge \Psi_2$, where $\Psi_1 \equiv a[T1] \leq 0 \wedge a[T2] \leq 0 \wedge a[T3] \leq 0$ and $\Psi_2 \equiv b[T1] \leq 2 \wedge b[T2] \leq 2 \wedge b[T3] \leq 2$. As a result, we manipulate Ψ by negating either Ψ_1 or Ψ_2 to compute unexplored symbolic states.

4.2.2 Prioritizing symbolic states and warps

In Algorithm 1, the procedure `InsertWithPriority` is used to create an ordered list of symbolic states (*cf.* the component "Program path explorer" in Figure 4). This ordering is guided by the static analyzer and it aims to expose as many performance issues as possible in a given time budget for testing.

To formalize the concept, let us assume two unexplored symbolic states $\Psi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$ and $\Psi' \equiv \psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_n$ for warp w. The set of threads in warp w is captured by $\mathcal{T}_w$. We write $\Psi \preccurlyeq_d \Psi'$ ($\Psi \preccurlyeq_i \Psi'$) if Ψ' has priority over Ψ in terms of generating global-memory interferences *directly* (*indirectly*). Finally assume that b_Ψ ($b_{\Psi'}$) is the control dependence edge in the CDG (*cf.* Figure 3(d)) capturing the symbolic condition ψ_m (ψ'_n).

To expose DRAM-interferences directly, we choose a symbolic state which is likely to lead to the maximum number of unexplored DRAM requests. Therefore, we prioritize Ψ and Ψ' as follows.

$$\Psi \preccurlyeq_d \Psi' \Leftrightarrow \sum_{t \in \mathcal{T}_w} |R_{mem}(b_\Psi, t)| \leq \sum_{t \in \mathcal{T}_w} |R_{mem}(b_{\Psi'}, t)| \quad (3)$$

$$R_{mem}(e, t) = \{m \in \mathcal{R}_t \mid e \rightsquigarrow m \wedge Chmc(m, t) \neq AH\}$$

where $\mathcal{R}_t$ is the set of all memory references by thread t and $Chmc(m, t)$ is the AH/AM/NC categorization of m in thread t (*cf.* Section 4.1.1). The notation $e \rightsquigarrow m$ captures that memory reference m is reachable from CDG edge e.

To systematically generate cache conflicts, we choose a symbolic state for warp w that may potentially lead to the maximum and unexplored cache conflicts to other warps. In particular, we prioritize Ψ and Ψ' as follows.

$$\Psi \preccurlyeq_i \Psi' \Leftrightarrow |C_{mem}(b_\Psi, w)| \leq |C_{mem}(b_{\Psi'}, w)| \quad (4)$$

$$C_{mem}(e, w) = \{m \in \mathcal{R}_w \mid e \rightsquigarrow m \wedge \mathit{Intf}(m, \mathcal{W} \setminus \{w\})\}$$

where $\mathcal{W}$ is the set of all warps, $\mathcal{R}_w$ is the set of all memory references by warp w and $\mathit{Intf}(m, W)$ captures whether m may conflict with any memory block accessed in the set of warps W. Once the symbolic states in individual warps are prioritized, we walk through the prioritized list $unex[w]$ for each warp w. As an outcome, we create a feasible symbolic expression Ω and generate a concrete input τ satisfying Ω to execute the entire GPGPU kernel.

To systematically select schedules, we prioritize warps during the execution (*cf.* the component "Schedule explorer" in Figure 4). In particular, when a warp is blocked during the execution (*e.g.* due to DRAM transaction or barrier synchronization), we issue a *ready warp* that is likely to lead to memory-performance issue. Such a priority function is guided both by the information computed during static analysis and the information available at runtime.

Let us consider two warps w_x and w_y which are ready to execute from program location p_x and p_y, respectively. We say $p_x \preccurlyeq_d p_y$ ($p_x \preccurlyeq_i p_y$), if warp w_y is given priority over warp w_x to generate DRAM interferences *directly* (*indirectly*). If we want to generate DRAM congestion, we attempt to create as many DRAM bank conflicts as possible. We assume that $B_1, B_2, \ldots, B_s$ are the different DRAM banks of global-memory. Besides, we use the notation $\mathit{Req}(B_i)$, which *dynamically* captures the set of outstanding requests in memory bank B_i. To create global-memory bank conflicts, we try to maximize the potential DRAM accesses to the *busy* banks (*i.e.* DRAM banks with outstanding requests). In particular, if a warp w executes from program location p, we compute the potential conflicts $\mathcal{G}_{B_i}(p, w)$ to memory bank B_i before w is blocked. Assuming $\mu(m)$ captures the global-memory bank where m is mapped, $\mathcal{G}_{B_i}(p, w)$ is formally defined as follows:

$$\begin{aligned}\mathcal{G}_{B_i}(p, w) = \{ \quad & m \quad \notin \mathit{Req}(B_i) \mid \mu(m) = B_i \wedge |\mathit{Req}(B_i)| \neq 0 \\ & \wedge \quad \exists t \in \mathcal{T}_w.\, m \in \mathit{Conflict}(p, t) \\ & \wedge \quad \mathit{Chmc}(m, t) \neq AH\} \end{aligned} \quad (5)$$

where $\mathit{Conflict}(p, t)$ denotes the *cache-conflict-analysis* fixed-point at p for thread t (*cf.* Section 4.1.3). The priority function $\preccurlyeq_d$ attempts to maximize the bank conflicts by issuing global-memory requests to the *busy* banks and it is formally defined as follows.

$$p_x \preccurlyeq_d p_y \Leftrightarrow \sum_{i=1}^{s} |\mathcal{G}_{B_i}(p_x, w_x)| \leq \sum_{i=1}^{s} |\mathcal{G}_{B_i}(p_y, w_y)| \quad (6)$$

Finally, let us consider prioritizing warps to create cache pollution. We assume that $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_q$ are the different cache sets and $\mathcal{C}(\mathcal{S}_i)$ *dynamically* captures the content of cache set $\mathcal{S}_i$. We use the notation $m \mapsto \mathcal{S}_i$, if memory block m is mapped to cache set $\mathcal{S}_i$. If a warp w executes from program location p, we compute the potential cache conflict $\mathcal{H}_{\mathcal{S}_i}(p, w)$ to cache set $\mathcal{S}_i$ before w is blocked. $\mathcal{H}_{\mathcal{S}_i}(p, w)$ is formally defined as follows.

$$\begin{aligned}\mathcal{H}_{\mathcal{S}_i}(p, w) = \{ \quad & m \quad \notin \mathcal{C}(\mathcal{S}_i) \mid \mathit{Reuse}(\mathcal{S}_i, \overline{w}) \cap \mathcal{C}(\mathcal{S}_i) \neq \phi \\ & \wedge \quad \exists t \in \mathcal{T}_w.\, m \in \mathit{Conflict}(p, t) \\ & \wedge \quad m \mapsto \mathcal{S}_i\} \end{aligned} \quad (7)$$

$\mathit{Reuse}(\mathcal{S}_i, \overline{w})$ captures the potentially reused cache content from set $\mathcal{S}_i$ by warps other than w. Note that $\mathit{Reuse}(\mathcal{S}_i, \overline{w})$ can be computed from the cache reuse information (*cf.* Section 4.1.2). Intuitively, we attempt to create cache pollution by replacing the reused cache content of waiting warps as much as possible. Therefore, we can now formally define the priority function $\preccurlyeq_i$ as follows.

$$p_x \preccurlyeq_i p_y \Leftrightarrow \sum_{i=1}^{q} |\mathcal{H}_{\mathcal{S}_i}(p_x, w_x)| \leq \sum_{i=1}^{q} |\mathcal{H}_{\mathcal{S}_i}(p_y, w_y)| \quad (8)$$

In our evaluation, we observed that the partial order $\preccurlyeq_i$ is more effective in exposing memory-performance problems compared to the partial order $\preccurlyeq_d$. This is primarily due to the small cache sizes configured for embedded GPUs. Therefore, in our framework, the partial order $\preccurlyeq_i$ is given priority over the partial order $\preccurlyeq_d$.

4.3 Monitoring the execution of a GPU kernel

In Algorithm 1, procedure `ExecuteAndMonitor` (*cf.* line 44 in Algorithm 1) monitors the execution of a GPU kernel, records the overhead due to DRAM congestion or on-chip cache conflicts and computes unexplored symbolic states for subsequent executions. In Figure 4, this component is shown via the label "Execution monitor". It is important to note that for each warp w, the unexplored symbolic states are generated and prioritized in the list $unex[w]$ using the exactly same methodology as shown in lines 9-27 of Algorithm 1. Therefore, in the following, we shall only discuss the monitoring process to record overhead in a specific execution. If $\mathcal{R}_w$ captures the set of all dynamic memory references by warp w and $\mathcal{T}_w$ denotes the set of all threads contained in w, we compute the total overhead $\mathcal{O}_w$ as follows.

$$\mathcal{O}_w = \sum_{m \in \mathcal{R}_w} \begin{cases} T_m - LAT_{hit}, \text{if } \forall t \in \mathcal{T}_w.\, \mathit{Chmc}(m, t) = AH; \\ \max(0, T_m - T_{round}), \text{otherwise;} \end{cases} \quad (9)$$

In Equation 9, T_m is the time taken to complete the memory request, T_{round} captures the total round-trip time to DRAM in the absence of DRAM congestion or DRAM bank conflicts and LAT_{hit} is the cache hit latency. Two cases in Equation 9 capture the scenarios in Figure 2(b) and Figure 2(a), respectively.

Handling synchronization points and program exit In a GPGPU program, different threads within the same thread block may synchronize via barriers. Besides, we consider the exit location of the GPU kernel to be an *implicit barrier*. Since a thread block may contain many warps, a specific warp might be delayed by other warps at the synchronization barrier. Since we specifically aim to detect memory-performance issues, our computed overhead (*i.e.* $\mathcal{O}_w$ for warp w) requires adjustment at these synchronization barriers. Figures 5(a)-(b) show two different scenarios when warps $w1$ and $w2$ suffer additional delay due to inter-warp interferences and synchronize at a barrier. Specifically, the adjustment depends on the timing behaviours of warps in the absence of inter-warp interferences.

To formalize, let us consider n warps $\{w_1, w_2, \ldots, w_n\}$ that need to synchronize at a barrier location. Besides, assume that warp w_i takes time T_{w_i} to reach the barrier location and $\mathcal{O}_{w_i}$ is the additional overhead (*cf.* Equation 9) suffered by w_i before reaching the barrier location. Finally, consider two warps w_m and $w_{m'}$ satisfying the following constraints.

$$T_{w_m} \geq \max_{i=1\ldots n} T_{w_i};\ T_{w_{m'}} \geq \max_{i=1\ldots n} (T_{w_i} - \mathcal{O}_{w_i}) \quad (10)$$

Therefore, w_m ($w_{m'}$) is the final warp to reach the barrier location in the presence (absence) of inter-warp interferences. The adjustment of the overhead can now be formally defined as follows.

$$\mathcal{O}_{w_i} = \begin{cases} \mathcal{O}_{w_{m'}} + T_{w_m} - T_{w_{m'}}, \\ \text{if } \max\limits_{j=1\ldots n} (T_{w_j} - \mathcal{O}_{w_j}) > T_{w_m} - \mathcal{O}_{w_m}; \\ \\ \mathcal{O}_{w_m}, \text{ otherwise.} \end{cases} \quad (11)$$

The first case in Equation 11 captures the scenario in Figure 5(a), whereas the second case captures the scenario in Figure 5(b).

Test suite property Upon the termination of Algorithm 1, we produce a set of test cases (the component "Test suite" in Fig-

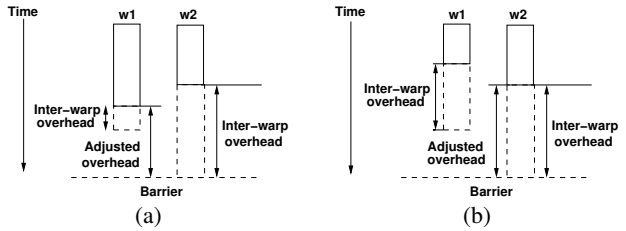

Figure 5. Adjusting overhead at a barrier when $w2$ is the final warp to execute the barrier and in the absence of inter-warp interferences (a) $w1$ executes longer, (b) $w2$ executes longer

ure 4) ranked by the magnitude of detected memory overhead (*i.e.* $\mathcal{O}_w$ in Equation 9). Each test case captures a potential memory-performance problem and it is a triplet of the form $\langle \Omega, \mathcal{SH}, \xi \rangle$. In a test case, Ω is a symbolic formula that captures the set of inputs that lead to a memory-performance problem, $\mathcal{SH}$ is a graph that captures the exact schedule during the respective execution and ξ is the set of memory interferences (*i.e.* DRAM congestion or on-chip cache conflicts) observed during the respective execution. A concrete input θ can be obtained by solving the formula Ω. The performance problem can be replicated by executing the program with θ and strictly enforcing the schedule $\mathcal{SH}$.

Since we aim to uncover memory-performance bottlenecks, the coverage of our test suite is provided via the set of explored memory references in each thread. However, we exclude memory references which are categorized AH during the static analysis and do not face any *inter-thread* cache conflict. This is because, such memory references do not generate any DRAM transaction. Additionally, our generated test suite satisfies the following property.

PROPERTY 4.1. *Consider two test cases* $\langle \Psi_{w_1} \wedge \Psi_{w_2} \wedge \ldots \wedge \Psi_{w_n}, \mathcal{SH}, \xi \rangle$ *and* $\langle \Psi'_{w_1} \wedge \Psi'_{w_2} \wedge \ldots \wedge \Psi'_{w_n}, \mathcal{SH}', \xi' \rangle \in \mathcal{T}$. *For any* $i \in [1, n]$, *we have* $\Psi_{w_i} \neq \Psi'_{w_i}$.

Intuitively, Property 4.1 states that in each warp, any symbolic state is explored *at most once* during the test generation process.

5. Experimental evaluation

Experimental setup Our implementation is based on the GPGPU-Sim simulator [8] and the GKLEE symbolic virtual machine [22]. We use the `nvcc` compiler to compile GPU kernels into CUDA-compliant binaries. GPGPU-Sim is used to extract PTX-level[2] control flow graphs (CFG) from CUDA binaries. These CFGs are fed as inputs to our static analyzer. We execute each kernel using GPGPU-Sim, while systematically generating warp schedules *on-the-fly* (*cf.* Section 4.2.2). To compute the symbolic state for each execution, we replay the respective traces using GKLEE. To generate concrete inputs from a symbolic formula, we use the STP constraint solver [5]. Our implementation is *completely automated* and it does not require any manual interventions. In our evaluation, we have used GPU kernels from CUDA SDK [2] (*cf.* Table 1). As our aim is to uncover memory-performance issues, we choose kernels which have substantial data accesses. Table 1 reports the salient features of all the evaluated CUDA kernels and the coverage of potential DRAM access locations obtained by GUPT. Finally, since embedded GPUs are primarily designed for low power consumption, they have small caches. Therefore, we configure the GPU platform (*cf.* Table 2) with 2-way, 4 KB instruction and data caches (cache sizes are similar to the configuration of the Vivante GC2000 embedded GPU [23]).

[2] Parallel Thread Execution (PTX) is a target-independent intermediate language used in CUDA programming environment

Program	Lines of code	Input size	Threads / block	Coverage
`bitonic_sort`	145	1 KB	256	100%
`convolution column`	191	32 KB	128	100%
`convolution_row`	188	32 KB	64	100%
`histogram`	248	64 KB	64	100%
`matmult`	151	3840 bytes + 3840 bytes	64	100%
`scan_b`	225	4 KB	1024	100%
`scan_w`	177	4 KB	1024	100%
`scan_naive`	158	4 KB	1024	100%
`transpose`	144	16 KB	256	100%
`transpose_naive`	145	16 KB	256	100%
`scalar_product`	141	4 KB + 4 KB	256	100%
`BFS`	218	nodes = 4096 edges = 28675	256	98%
`Fast-Walsh`	234	8 KB	512	100%
`Clock`	71	2 KB	256	100%

Table 1. Evaluated CUDA SDK kernels and obtained coverage

SIMD cores	Four cores, SIMD width = 8
Size of register file/SIMD core	2 KB
Instruction cache/SIMD core	2-way, 4 KB. Replacement policy = LRU line size = 64 bytes
Data cache/SIMD core	2-way, 4 KB. Replacement policy = LRU line size = 64 bytes
Texture cache/SIMD core	Not present
Constant cache/SIMD core	Not present
Shared memory/SIMD core	Not present
Minimum DRAM latency	100 compute core cycles
DRAM model	1 memory controller, bank size = 8 KB, total DRAM banks = 4, scheduling policy = *first-in-first-out* (FIFO)

Table 2. Micro-architectural configuration of the GPU platform

Key result We demonstrate the key results from our test generation framework via Figure 6. Figure 6 shows the number of memory-performance issues detected in `bitonic_sort` with respect to time. The kernel `bitonic_sort` has an exponential number of input-dependent paths. As a result, it is potentially infeasible to test the application for all possible execution paths and scheduling decisions. To evaluate the efficacy of our test generation framework, we compare with two different strategies, namely *pure random testing* and *symbolic random testing*. In pure random testing, we randomly generate inputs to execute the kernel. Moreover, when a warp is blocked, we randomly schedule a ready warp for execution. Symbolic random testing is similar to our test generation framework, except that *we do not use any guidance from our static analyzer*. Therefore, we randomly choose an unexplored symbolic state and we randomly schedule ready warps for execution. But, as opposed to pure random testing, symbolic random testing guarantees to execute different program paths for each generated test input. In Figure 6, we limit the maximum time budget for testing to 10 minutes. We make the following crucial observations from Figure 6. Since test inputs and schedules are generated randomly in pure random testing, the generated execution scenarios may not provide a good coverage of the potential performance problems. As a result, pure random testing reports relatively low number of performance issues. In symbolic random testing, program inputs are generated systematically by manipulating the symbolic states (*cf.* Algorithm 1). However, in the presence of a huge number of symbolic states and scheduling decisions, it is important to generate relevant execution scenarios as early as possible. We accomplish this by guiding the test generation via a static analyzer. As observed from Figure 6, we detect memory-performance problems early during the test generation, compared to symbolic random testing. When run long enough, the efficacy of symbolic random testing is slowly growing.

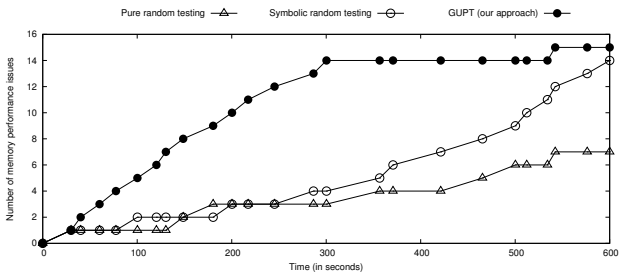

Figure 6. No. of detected memory-performance issues *w.r.t.* time

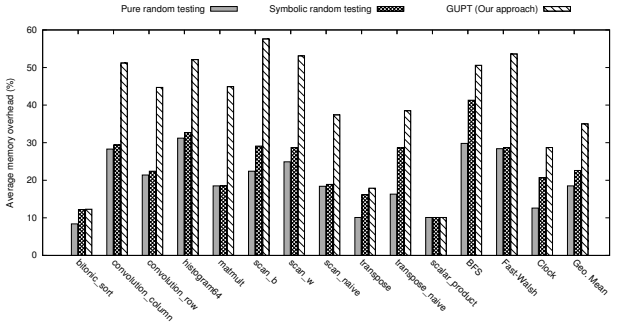

Figure 7. Evaluation with CUDA SDK kernels

Evaluation Figure 7 outlines the results of our experiments. For our experiments, we limit the time budget for testing as ten minutes. Recall that our framework records an overhead $\mathcal{O}$ (*cf.* Equation 9) to capture the amount of DRAM congestion or on-chip cache conflicts in a particular execution. Given $\mathcal{O} > 0$, we capture the magnitude of this overhead using the ratio $\frac{\mathcal{O}}{\mathcal{E}}$, where $\mathcal{E}$ is the total execution time and report the geometric mean of this ratio (*cf.* "Average memory overhead" in Figure 7) over different executions. We also use the average memory overhead to compare different test generation methodologies. This is because a high ratio indicates a substantial amount of DRAM congestion or on-chip cache conflicts and such memory-performance issues should be fixed in order to efficiently execute a kernel. As shown in Figure 7, GUPT significantly outperforms pure random testing. This is expected, as pure random testing does not have any knowledge about program paths and micro-architectural states (*e.g.* caches). Symbolic random testing overall performs better than pure random testing, except for GPGPU programs that have very few *input-dependent* paths (*e.g.* `matmult`). However, since symbolic random testing does not utilize any knowledge about micro-architectural states (as computed by our static analyzer), the detected overhead is smaller compared to the same detected by GUPT. Besides, we observed that most of the performance problems are detected early during our test generation (*e.g.* in Figure 6). Therefore, we believe that the combination of static analysis and symbolic testing provides a promising direction for performance testing GPU kernels.

Usage of the framework to compare different implementations We can use our test generation framework to compare different implementations of the same algorithm. For instance, we evaluate different implementations of the *parallel prefix sum* algorithm (`scan_naive`, `scan_w` and `scan_b` in Table 1) from CUDA SDK. `scan_naive` is algorithmically inefficient, using $O(n \cdot logn)$ add operations, whereas the other two kernels (*i.e.* `scan_w` and `scan_b`) use only $O(n)$ add operations. However, by systematically generating execution scenarios, we observed that both `scan_w` and `scan_b` may execute *longer* compared to `scan_naive` for certain scenarios. From the generated test report, we found that both `scan_b` and `scan_w` face around 20% more memory-overhead compared to `scan_naive` and this overhead affects the overall execution time. Therefore, we conclude that an algorithmically more efficient code is *insufficient* to accomplish better performance in complex GPU architecture. This is primarily due to the reason that such corner-scenarios are hidden via the programming layer and therefore, such scenarios are potentially impossible for the developer to detect manually. These performance problems can be highlighted to the developer via our proposed methodology.

Program	Execution cycle		Refactoring method
	Original	Refactored	
`Fast-Walsh`	795007	494474	cache locking
`transpose_naive`	3810	3251	changing cache layout
`scan_w`	16457	15175	changing cache layout
`matmult`	25652	10686	changing cache layout + cache locking
`BFS`	7827	7204	changing DRAM layout

Table 3. Refactoring GPGPU programs based on test results

Usage of the framework to optimize GPGPU programs Finally, we show the usage of our framework to generate efficient GPGPU code. Table 3 summarizes our evaluation.

The kernel `Fast-Walsh` operates on input data from *shared memory*. As embedded GPUs are targeted for power-constrained mobile devices, the availability of an on-chip shared memory is often restricted or such an on-chip memory is unavailable altogether in embedded GPUs [1, 6]. As a result, our framework identified a number of scenarios generating substantial DRAM traffic and it highlighted the accesses to the input data facing inter-warp cache conflicts. To reduce these cache conflicts, we *locked* the portion of input data that can fit into the L1 data cache (4 KB). This leads all other data accesses to *cache misses*. However, since the input data was reused substantially compared to any other variable, we observed around 30% reduction in overall execution time.

We studied the naive implementation of matrix-transpose (*i.e.* `transpose_naive`) and the kernel `scan_w` to improve their memory-performance. For these kernels, our framework generates execution scenarios that highlight inter-warp cache conflicts between two variables (`idata`,`odata` for `transpose_naive` and `ai`,`bi` for `scan_w`). To reduce such cache conflicts, we manually change the layout of `idata` and `odata` (`ai` and `bi`, respectively) so that they map to contiguous locations in the data cache. These changes improve the overall execution time by 14.6% (7.8%) for `transpose_naive` (`scan_w`). However, it is worthwhile to point that the kernel `transpose_naive` also suffers from *uncoalesced* memory accesses, which we ignore in our framework.

In `matmult`, we observed the efficacy of both cache locking and *layout* changing. Our framework highlighted inter-warp cache conflicts between matrix blocks As and Bs. We changed the memory layout of As and Bs to map them into contiguous locations in the L1 data cache. This modification did marginal improvements to the execution time, as our framework highlighted execution scenarios where As and Bs conflict with other data variables in the kernel. To resolve this, we locked the contents of As and Bs before they are reused and unlocked them at the exit of each GPU thread. These changes reduce the overall execution time by 58.3%.

In our experiments with the `BFS` kernel, our framework highlighted a substantial DRAM bank conflicts between two variables `g_graph_nodes` and `g_cost`. We change the DRAM addressing layout of these two variables to avoid DRAM bank conflicts. We obtained an improvement of around 8% via this modification.

In summary, preceding examples motivate that our framework can be used by a developer to improve the overall performance of a GPGPU program, manually or automatically via a compiler.

6. Related work

To understand the performance behaviour of a program, a significant research has been performed on *program profiling* over the last few decades [9, 13, 20, 21, 26]. Such techniques require inputs to run a program. Additionally, the works in [13, 26] extend traditional profiling techniques to compute a *performance pattern* over different inputs via empirical models. Our work complements program profiling by systematically finding performance-stressing test inputs. Besides, our work does not suffer from approximations incurred in empirical models and we also provide a coverage of memory-access locations to qualitatively evaluate our test suite. The performance characteristics of mainstream GPUs have recently been studied via analytical models [7, 16]. Our goal is orthogonal in the sense that we automatically generate test cases to uncover poor memory-performance. Therefore, our work has a significant testing flavour compared to the approaches proposed in [7, 16]. Besides, our framework does not rely on analytical models and detect memory-performance issues by monitoring program executions.

Existing works on *concolic testing* have mostly concentrated on sequential applications to detect functionality bugs [11, 15] and more recently, to detect poor cache performance [10]. Effective testing of parallel software has also emerged to be a critical problem. For parallel applications, an additional challenge is to systematically detect thread scheduling patterns that may lead to software bugs [14, 24]. The choice of such scheduling patterns has primarily focused on functionality bugs. In contrast, we aim to detect performance problems. Finally, the work in [22] has investigated the detection of functionality bugs and a few performance bugs (*e.g. shared memory bank conflicts* and *uncoalesced memory accesses*) for GPGPU applications. All such bugs can be detected by examining a specific schedule, as called *canonical schedule* in [22]. Since different thread scheduling patterns may generate different memory traffic, memory performance problems might not be exposed via the canonical schedule. To detect such performance problems, it is critical to select appropriate thread schedule and test inputs. We accomplish this by a test generation strategy which is guided via static analysis. Besides, we only report memory-performance issues that affect the overall *execution time*.

In summary, our work extends the formal foundation of software testing via a test generation framework that systematically generates test inputs and thread schedules to detect memory-performance issues in embedded GPGPU applications, and in general, for embedded parallel applications.

7. Conclusion

In this paper, we have proposed GUPT, a fully automated software-testing framework to uncover memory-performance issues in embedded GPUs. We leverage on dynamic test generation and systematically guide the test generation process via a static analyzer. Our experiments with several GPGPU programs suggest that we can uncover a number of memory-performance issues early during the test generation. We show the usage of our framework in comparing different implementations of the same functionality and in improving the performance of applications for embedded GPUs.

Since our approach is based on dynamic test generation, we do not generate *false alarms*, but we might miss some memory-performance issues. Besides, our test generation primarily focuses on exposing DRAM and cache interferences and ignores other performance problems due to warp interferences, such as long waiting-time at synchronization barriers.

In the future, we plan to extend our framework to *automatically* localize the root cause (*i.e.* performance debugging) of performance issues. Subsequently, we plan to automatically suggest refactoring techniques to resolve such performance issues.

References

[1] ARM mali T600 series GPU OpenCL. http://infocenter.arm.com/help/topic/com.arm.doc.dui0538e/DUI0538E_mali_t600_opencl_dg.pdf.

[2] CUDA SDK. https://developer.nvidia.com/cuda-downloads.

[3] CUDA toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] LLVM compiler infrastructure. http://llvm.org/.

[5] STP constraint solver. https://sites.google.com/site/stpfastprover/.

[6] Vivante graphics core. http://www.vivantecorp.com/index.php/en/technology/gpgpu.html.

[7] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W-m W. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, 2010.

[8] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, 2009.

[9] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.

[10] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Static analysis driven cache performance testing. In *RTSS*, 2013.

[11] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[12] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In *RTSS*, 2009.

[13] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, 2012.

[14] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *OOPSLA*, 2013.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[16] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[17] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A Qos-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *DAC*, 2012.

[18] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ASPLOS*, 2013.

[19] H. Jooybar, W. WL. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt. GPUDet: A deterministic GPU architecture. In *ASPLOS*, 2013.

[20] S. Lagraa, A. Termier, and F. Pétrot. Scalability bottlenecks discovery in MPSoC platforms using data mining on simulation traces. In *DATE*, 2014.

[21] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[22] 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. http://www.cs.utah.edu/formal_verification/GKLEE/.

[23] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng. General purpose computing on low-power embedded GPUs: Has it come of age? In *SAMOS*, 2013.

[24] S. Nagarakatte, S. Burckhardt, M. MK. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, 2012.

[25] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.

[26] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, 2012.