Systematic Detection of Memory Related Performance Bottlenecks in GPGPU Programs

Adrian Horga[1] Sudipta Chattopadhyay[2] Petru Eles[1] Zebo Peng[1]

[1]Linköping University [2]Centre for IT-Security, Privacy and Accountability

adrian.horga@liu.se, sudiptac@st.cs.uni-saarland.de, {petru.eles, zebo.peng}@liu.se

Abstract

Graphics processing units (GPUs) pose an attractive choice for designing high-performance and energy-efficient software systems. This is because GPUs are capable of executing massively parallel applications. However, the performance of GPUs is limited by the contention in memory subsystems, often resulting in substantial delays and effectively reducing the parallelism. In this paper, we propose GRAB, an automated debugger to aid the development of efficient GPU kernels. GRAB systematically detects, classifies and discovers the root causes of memory-performance bottlenecks in GPUs. We have implemented GRAB and evaluated it with several open-source GPU kernels, including two real-life case studies. We show the usage of GRAB through improvement of GPU kernels on a *real NVIDIA Tegra K1 hardware* – a widely used GPU for mobile and handheld devices. The guidance obtained from GRAB leads to an overall improvement of up to 64%.

Keywords: Performance debugging, GPGPU, Caches

i

1. Introduction

Since the inception of high-performance and energy-efficient platforms (*e.g.* multi-cores, graphics processing units), it has become critical to design applications that efficiently utilize the full potential of such execution platforms. However, it is notoriously difficult to build correct and efficient software in these platforms. In particular, embedded software are often constrained via timing-related constraints or limited battery power. In order to build efficient applications on emerging computing platforms, we envision a framework that aids, in particular, the validation of performance. Concretely, our goal is to systematically locate performance bottlenecks, highlighting their root causes and suggesting appropriate fixes to improve the performance, with a specific focus on GPUs.

In the last decade, mainstream [1] and embedded GPUs [2, 3] have gained popularity in designing non-graphics or general-purpose applications (as often called GPGPU applications). GPUs can execute massively parallel applications by spawning thousands of threads in parallel. Besides, GPUs offer high-level programming abstractions. Using such abstractions, a developer can write the code for one thread and can specify the number of threads to execute. The amount of parallel computations, during the execution, acts as the key factor in performance gain obtained from GPUs. The execution behavior, and hence, the performance of a GPGPU program, in turn, is critically influenced by the underlying architecture. However, the very complex architectural details of GPUs remain *completely hidden* from the developer. As a result, it is potentially impossible for a developer to discover and understand performance bottlenecks in GPUs. In particular, accesses to the global-memory (DRAM), in GPUs, are often several orders of magnitude slower than accessing on-chip memories withing GPU cores (*e.g.* caches, registers and shared memory). Due to the slow response time of DRAM, it is often likely that multiple threads wait for the DRAM at the same time. Therefore, even if GPU threads are functionally independent, they may create interference among each other due to shared resources such as DRAM. As a result, a significant number of accesses to the global-memory may lead to memory contention, leading to a substantial interference among GPU threads and poor overall execution time. Providing appropriate methodologies to localize and understand memory-related bottlenecks is the main contribution of this paper.

The state-of-the-art in performance debugging has long been *profiling* [4, 5, 6, 7, 8, 9]. Unfortunately, program profiling has several drawbacks in the context of highlighting performance problems. First of all, profiling highlights program locations where time is *spent* (*i.e.* hotspots), instead of locations where time is *wasted*. Secondly, performance bottlenecks might be ranked lower by profilers if they spend less time compared to the rest of the program. Finally, program profiling, by its very nature, does not isolate the *cause* of performance bottlenecks.

GPUs offer very limited on-chip memories [10], such as caches and registers. If a program variable cannot be allocated in registers, accessing the variable will generate a memory request. For a particular memory request, caches are searched first. If the requested memory block is not found in the cache, it has to be fetched from the DRAM. Therefore, for a limited cache size, a large number of threads can generate substantial interference in the cache, which may dramatically increase the number of slow DRAM accesses. This, in turn, may lead to a scenario, where a large number of threads would wait for their requested memory blocks.

In this paper, we propose GRAB, a systematic methodology that takes a straightforward implementation in GPU (from the respective implementation in CPU) and gradually isolates the causes of memory-related bottlenecks in the implementation. In particular, the number of accesses to the global-memory might increase substantially due to the complex interactions among threads or due to an excessive number of threads in the GPGPU program. GRAB locates the cause of such interference and presents a bug report to the developer. This bug report can be used to systematically transform the straightforward GPU implementation to more efficient versions. Thus, our framework aids inexperienced GPU programmers to write more efficient GPU code and subsequently guide them to become better GPU programmers.

In order to detect the interference in the memory subsystems, we face several technical challenges. Since the execution-pattern of threads and the architectural details of a GPU remain hidden to the developer, it is potentially impossible to detect interference in the memory subsystems, solely by inspecting a GPGPU program. To resolve such challenges, we execute the GPGPGU program in an environment which models the specific architectural details of the underlying GPU. Besides, we execute the GPGPU program in a contrived fashion. In particular, we first carefully intercept all the memory requests (*i.e.* accesses that go through caches and global-memory) issued by the program, during its execution. For each memory request, we manipulate two different execution traces – an *original trace* and a *golden trace*. For the original trace, we respect the execution pattern in the given GPU architecture. However, we manipulate the golden trace in such a fashion that the execution proceeds *without any interference among threads*. The golden trace acts as an *ideal reference* to locate all the memory requests, which are affected due to the interference across threads.

We use the original and golden execution traces to localize the cause of interference in the memory subsystems. In particular, for each memory request, we check whether the original and the golden trace are manipulated differently. If a difference was observed, it was caused due to the interference among threads. Subsequently, we process the execution trace backwards to discover the root cause of such a difference. Once the root cause was found, we appropriately record this information, which avoids searching for the same root cause again. This potentially reduces the debugging time substantially, when a large amount of interference among threads is caused due to a few instructions.

Our debugging process continues until the given GPGPU program finishes execution. At the end of the execution, we have discovered and localized all the memory requests, which are affected due to the interference across threads. We use this information to generate a bug report for the developer. The primary purpose of this bug report is to classify and prioritize the observed interference in the memory subsystems and their respective causes. We classify each entry in the bug report according to its potential fix. For instance, we group all global-memory requests, which might be reduced by improving the cache management, such as by changing the memory-access pattern or memory layout. Moreover, the bug report also prioritizes root causes discovered for interference in the memory subsystems. This helps to focus on the more important performance-bottlenecks first. Therefore, the generated bug report concisely reports the set of memory bottlenecks, which can be used to understand and improve the memory performance of GPGPU programs.

Contributions. In summary, we designed and developed a systematic debugging methodology to write efficient GPGPU programs on embedded computing platforms. Our proposed methodology discovers interference in the memory subsystems *on-the-fly* and by manipulating two different execution traces simultaneously. In particular, one of the execution traces is used as a reference to localize the interference across threads. GRAB is primarily used to generate a bug report for the developer. The bug report captures the necessary information to understand the memory performance problems of the GPGPU program and their potential solutions. Finally, the entire process to generate the bug report is *automatic* and it does not require any manual intervention. We have implemented our debugging framework using `GPGPU-Sim` [11], a cycle-accurate, open-source GPU simulator. We have evaluated our framework on several open-source GPU kernels, all available in [12]. *In particular, we show the usage of GRAB by improving the performance of GPGPU programs up to 36% on two real-life case studies, on an NVIDIA Tegra K1 GPU.*

2. System and Execution Model

Figure 1 demonstrates the GPU execution platform targeted by our debugging framework. The GPU contains a number of streaming multiprocessors (SM). All cores in an SM share an L1 cache. Additionally, an SM may contain a software-controlled shared memory. All memory blocks, which are not allocated into shared memory, need to be fetched from the slow global-memory (DRAM). The number of accesses to the DRAM can be reduced dramatically due to the presence of caches. It is worthwhile to mention that the architecture shown in Figure 1 is typical for both mainstream [1] and embedded GPUs [2]. Our debugging framework primarily targets to localize the global-memory accesses due to inter-thread interference. More specifically, we aim to detect the following scenarios:

- Since several threads in an SM share the cache, a memory block might be evicted from the L1 cache entirely due to inter-thread cache conflicts.

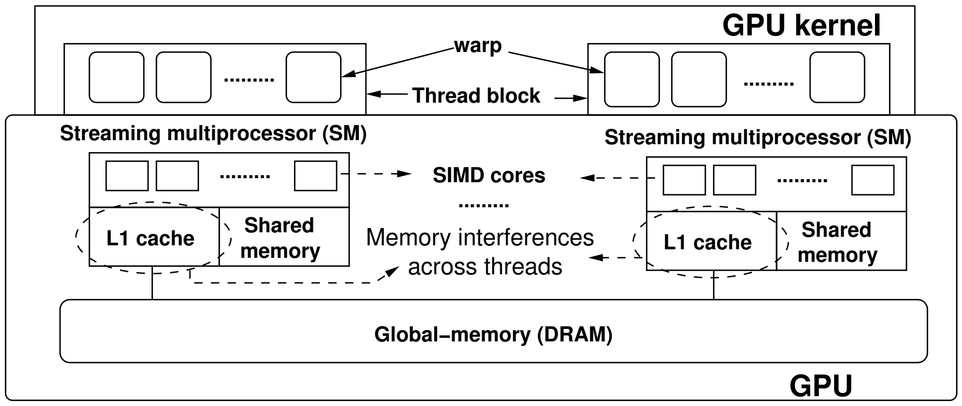


Figure 1: Execution model (SIMD stands for "single instruction multiple data")

- If a substantial number of threads in an SM share the cache, it may generate a significant amount of global-memory requests, due to an inadequate size of the L1 cache.

At present, our framework does not consider L2 caches, which are employed in some current-generation GPUs [1]. However, the extension of our debugging framework for multi-level caches is straight-forward and we consider such an extension for future developments.

The input to our debugging framework is an implementation of a GPU kernel using CUDA [13][1]. The execution unit of a CUDA kernel is a *warp*. The warp, in turn, belongs to a specific thread block. The user only specifies the number of thread blocks and the size of a thread block in the program. Warps are executed within the GPU via a vendor-specific scheduling policy, typically *unknown* to the developer. This leads to the potential impossibility for a developer to discover and understand interference across threads and locate performance bottlenecks in the memory subsystems.

Background on caches. Caches are employed to bridge the performance gap between the processing cores and the DRAM (*cf.* Figure 1). A cache can be described as a triplet $\langle \mathcal{A}, \mathcal{S}, \mathcal{L} \rangle$, where $\mathcal{A}$ is the associativity of the cache, $\mathcal{S}$ is the number of cache sets and $\mathcal{L}$ is the line size (in bytes). Each cache set can hold $\mathcal{A}$ cache lines, leading to a total cache size of $(\mathcal{A} \cdot \mathcal{S} \cdot \mathcal{L})$ bytes. When $\mathcal{A} = 1$, the respective caches are called to be *directly mapped*. Data is fetched into caches at the granularity of

[1]Our framework is also equally applicable for OpenCL applications.

line size ($\mathcal{L}$). The number of cache sets ($\mathcal{S}$) decides the location where a particular memory block would be placed in the cache. For instance, a memory block, starting at address M, is always mapped to the cache set $(M \; mod \; \mathcal{S})$. If two memory blocks M_1 and M_2 are mapped to the same cache set, we say that M_1 conflicts with M_2 in the cache and vice versa. Since each cache set can hold only $\mathcal{A}$ cache lines, a cache line needs to be replaced when the number of memory blocks mapping to a cache set exceeds $\mathcal{A}$. In order to accomplish this, a replacement policy is employed when $\mathcal{A} \geq 2$. Our framework works on any replacement policy but we primarily discuss two widely used policies – *least recently used* (LRU) and *first in first out* (FIFO). In the LRU policy, the memory block, that was not *accessed* for the longest period of time, is replaced from the cache to make room for other memory blocks. In the FIFO policy, the memory block, which is *residing* in the cache for the longest period of time, is replaced to make room for other blocks.

3. Overview

Challenges in debugging:. Detecting memory bottlenecks in GPGPU programs involves several technical challenges. One such key challenge is to obtain memory access information at the granularity of individual *threads*. Needless to say, such information is essential to detect thread-level interference. Unfortunately, current-generation GPUs do not provide adequate support to extract thread-level performance statistics during execution. In order to solve this problem, we execute and manipulate traces within a simulator `GPGPU-Sim` [11], which has been shown to model the performance of GPUs in a highly accurate fashion [11]. Besides, our evaluation also shows that utilizing traces obtained from `GPGPU-Sim` is a useful technique in practice, in order to improve the performance of GPGPU programs on real hardware.

An example. We illustrate the mechanism of our framework via a simple example. Let us assume that three threads $T1$, $T2$ and $T3$ are running in parallel and they are sharing a two-way set associative cache employing FIFO replacement policy. We presume the cache can hold at most four memory blocks, meaning that there exists only two different cache sets. All accesses that miss the cache, must fetch the respective memory block from the slow global-memory. Finally, we assume that $m1$, $m2$, $m1'$, $m2'$ and $m3'$ are all mapped to the cache set $\mathcal{S}_1$. For the sake of illustration, we shall consider a thread to be the execution unit. Our methodology is equally applicable for *warps*.

In Figure 2 - Figure 5, $T1$, $T2$ and $T3$ are threads running in parallel. Memory blocks accessed by each thread are shown in a sequence. For instance, thread $T2$ accesses memory blocks $m1'$, $m1'$ and $m2'$ in a sequence. The direction of the

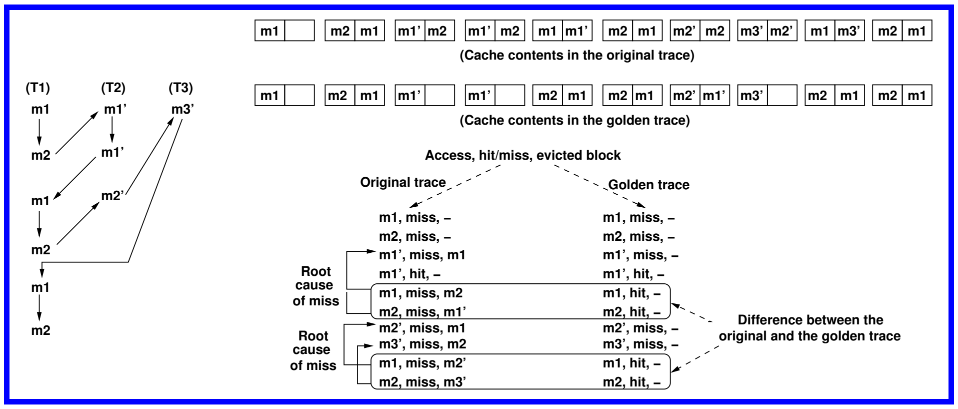


Figure 2: Performance debugging when all memory blocks are mapped to the same cache set

arrow (among $T1$, $T2$ and $T3$) captures the order in which the memory requests are scheduled. For each memory request, we *only* show the content of the cache set being accessed.

Figure 2 highlights the order, in which the memory requests are scheduled across threads. To discover interference in the memory subsystem, we execute the program in a contrived fashion. During the execution, we *intercept* all memory requests made by the program. The primary purpose of intercepting memory requests is to check whether the respective request leads to an access to the global-memory (*cf.* Figure 1). For each memory request during the execution, we manipulate two different traces – an *original* trace and a *golden* trace. The original trace records the cache hit/miss statistics for each memory request in the original execution. On the contrary, the golden trace captures the cache hit/miss statistics assuming *the absence of any inter-thread cache conflicts*. In other words, during the manipulation of the golden trace, we assume that each thread has its own copy of the cache. We emphasize that the golden trace cannot be produced simply by simulating each thread *in isolation*. This is due to the reason that the control flow of different threads might depend on each other (*e.g.* via updating and reading *shared* variables).

In Figure 2, we highlight the changes in cache contents both for the original trace and the golden trace (as labeled "original trace" and "golden trace"). For the sake of simplicity, we only show the cache content of the cache set being accessed. Since all memory blocks are initially mapped to cache set $\mathcal{S}_1$, Figure 2 captures the content of cache set $\mathcal{S}_1$ only. For instance, let us consider the first access to block $m1'$ by thread $T2$. If thread $T2$ would have a separate copy of the cache (*i.e.* in the golden trace), its first access to $m1'$ does not evict memory block $m1$. In Figure 2, we capture both traces via a sequence of triplets of the form $\langle access, hit/miss, evicted \rangle$, where $access$ captures the accessed memory block, $hit/miss$ captures whether the respective access was a cache hit and $evicted$ captures the memory block evicted due to $access$. Figure 2 highlights these two traces for our example. Note that the first access to $m1'$ (in the original trace) evicts memory block $m1$. Therefore, this information is captured via the triplet $\langle m1', miss, m1 \rangle$ in the original trace.

In Figure 2, we have also highlighted the difference between the original and the golden trace. For instance, non-first accesses to memory blocks $m1$ and $m2$ behave differently for the original trace and the golden trace. Recall that the golden trace is produced assuming that each thread has its own copy of the cache. Therefore, the difference between the original trace and the golden trace captures the scenario when memory blocks $m1$ and $m2$ suffer cache conflicts across threads. Our goal is to reduce such interference by highlighting their root causes. For instance, consider the triplet $\langle m2, miss, m1' \rangle$ in the original trace. We discover that the access to $m2$ is a miss and we check whether it has been evicted previously by another memory access. We can find that memory block $m2$ was evicted by $m1$, which, in turn was evicted by $m1'$. Therefore, the access to $m1'$ causes the cache miss for the access $\langle m2, miss, m1' \rangle$. In this way, we can obtain the root cause of all differences between the original and the golden trace, as captured in Figure 2. In our framework, we discover all root causes *on-the-fly*.

After finding all root causes, we rank them according to the number of interferences (*i.e.* cache misses) they cause. For instance, the first access to $m1'$ causes two interferences and accesses to $m2'$ and $m3'$ cause one interference each (*cf.* Figure 2). This ranking is presented to the developer along with the suggestions for possible fixes. In particular, our framework categorizes any memory request as follows:

- hit : The requested memory block was in the cache.
- $miss^*$: The requested memory block was not found in the cache and all cache lines were occupied by some memory block requested in the past.
- $miss$: The memory request cannot be categorized either as hit or as $miss^*$.

The categorization $miss^*$ aims to approximate capacity misses (*i.e.* cache misses due to the limited cache size). We note that true capacity misses are difficult to predict without analyzing all memory requests in the future. Since our debugging approach works *on-the-fly*, we use an approximation for capacity misses. This keeps the debugging time short and avoids analyzing long traces. For memory requests categorized $miss$, we observe that cache misses might be reduced by a different cache mapping. For instance, in Figure 2, all memory blocks are mapped to cache set $\mathcal{S}_1$. As a result, accesses to $m1$ and $m2$ were categorized $miss$. To improve the memory performance, we suggest a rectification that map memory blocks differently in the cache. More specifically, in Figure 2, the root cause $m1'$ is ranked higher than other root causes and it causes interference to memory blocks $m1$ and $m2$. Therefore, we suggest to relocate $m1'$, so that it is mapped into a different cache set than $m1$ and $m2$. More than one modification can be carried out before invoking our debugger, but, in this example, we show the invocation of our debugger after each single modification.

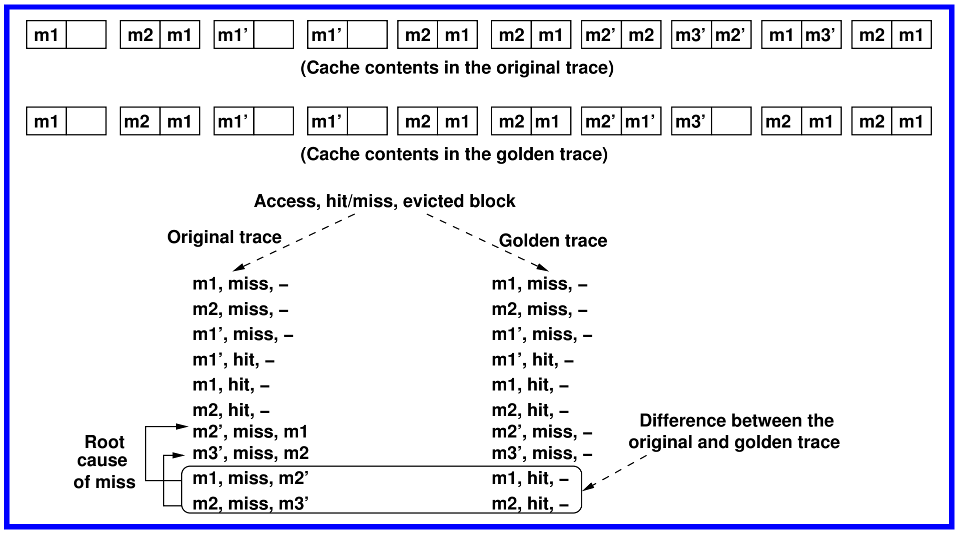


Figure 3: Performance debugging when $m1$, $m2$, $m2'$ and $m3'$ are mapped to the same cache set, but $m1'$ is moved to a different cache set. **Note:** The memory request order is the same as in Figure 2

Figure 3 captures the execution scenario after $m1'$ is mapped to $\mathcal{S}_2$. If the performance of the program in Figure 3 is acceptable, we can stop to make any further refinement. Otherwise, we can run our framework to discover the memory interference shown in Figure 3. Using the same procedure, we can highlight the root cause of such interference, as shown in Figure 3. In this case, we also discover that memory blocks $m1$ and $m2$ face interference due to $m2'$ and $m3'$, respectively. In this scenario, $m2'$ and $m3'$, each causes one interference and, hence, they are ranked equally. We can also discover that the cache set $\mathcal{S}_2$ is occupied only by $m1'$ requested in the past. Therefore, the possible fix was suggested again to improve the cache mapping. Since $m2'$ and $m3'$ are ranked equally, let us assume that we choose $m2'$ to be relocated in memory, so that $m2'$ is mapped to $\mathcal{S}_2$.

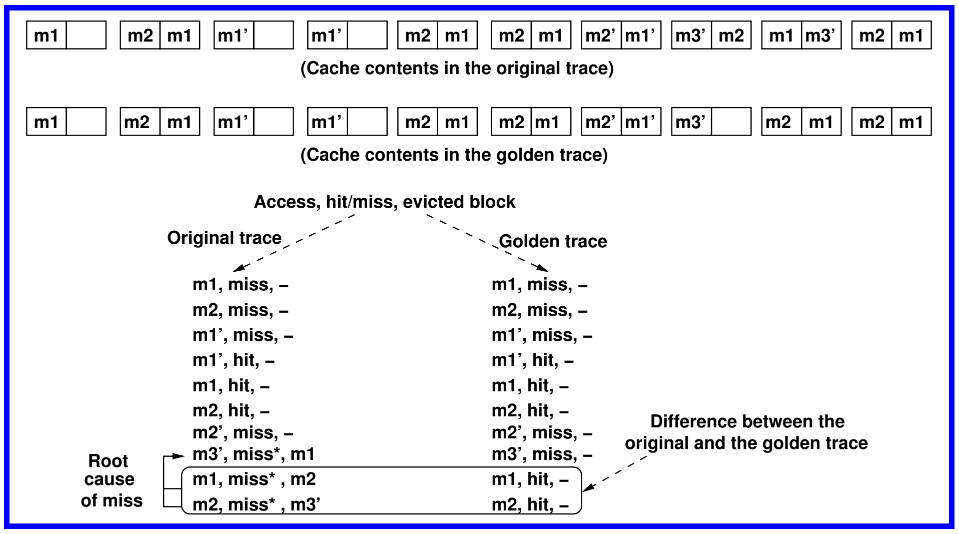


Figure 4: Performance debugging when $m1$, $m2$ and $m3'$ are mapped to the same cache set, but $m1'$ and $m2'$ are moved to a different cache set. **Note:** The memory request order is the same as in Figure 2

After the preceding modification, the execution scenario is captured via Figure 4. We can observe that the number of cache misses remains the same as before the modification. However, all differences between the original and the golden trace are caused by access to $m3'$. Besides, we note that for any observed difference, all cache lines were occupied by a memory block requested in the past. This is highlighted by "$miss$*" for accesses to $m1$ and $m2$.

One possible way to reduce such cache misses would be to merge thread $T1$ (which accesses $m1$ and $m2$) and $T3$ (which accesses $m3'$). It is important to note that reducing the number of threads reduces the amount of parallel computations and thus, may increase the overall execution time. Moreover, merging two parallel threads may require non-trivial changes in the source code and therefore in general, such changes should be performed interactively with the developer. Of course, if a change performing the merge between threads does not improve the overall performance, such a change could always be ignored by the developer.

The execution scenario after merging thread $T1$ and $T3$ are shown in Figure 5. In this scenario, we do not observe any memory interference across threads. However, we reduce the amount of concurrency by merging two threads. We believe that the decision to make such modifications should be left to the developer. In general, the number of threads and amount of memory contention pose a complex

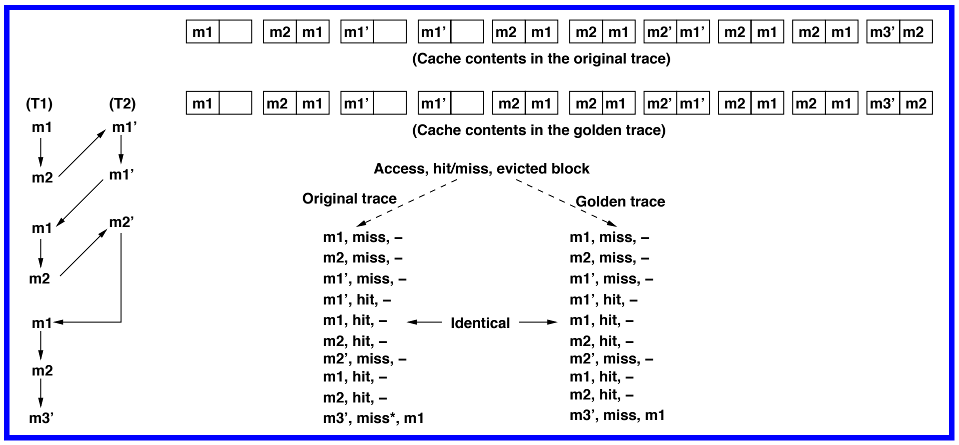


Figure 5: Performance debugging when merging threads $T1$ and $T3$ to reduce interference among threads

design trade-off and our goal is to help the developer to choose an appropriate implementation using our framework.

In summary, our debugging framework aims to interact with the developer by highlighting memory interference across threads and by suggesting their possible fixes. At any moment, such interference can be ignored by the developer when the obtained performance is acceptable. We emphasize that the primary goal of our framework is to present the developer a prioritized list of statements, which cause memory interference across threads and they might dramatically reduce the performance gain obtained from GPUs. Such a list can be used either as hints for optimizing compilers or as suggestions for manually modifying the GPGPU program.

Debugging framework. Figure 6 captures the workflow of GRAB. In this work, we do not focus on the test generation problem to expose memory bottlenecks in GPGPU programs. Therefore, we assume the presence of an existing test-suite, using which the GPU kernel can be executed. Such test suite can be generated via fuzzing the GPU kernel or via systematic testing [14]. Broadly, we perform the following operations to localize the cause of memory interference across threads.

- We monitor the execution of the GPU kernel (*execution monitor*) and intercept every memory request made by the GPU kernel. We accomplish this in a simulator, which models the architecture of the underlying GPU. We do not need to modify the GPGPU program to intercept its memory requests,

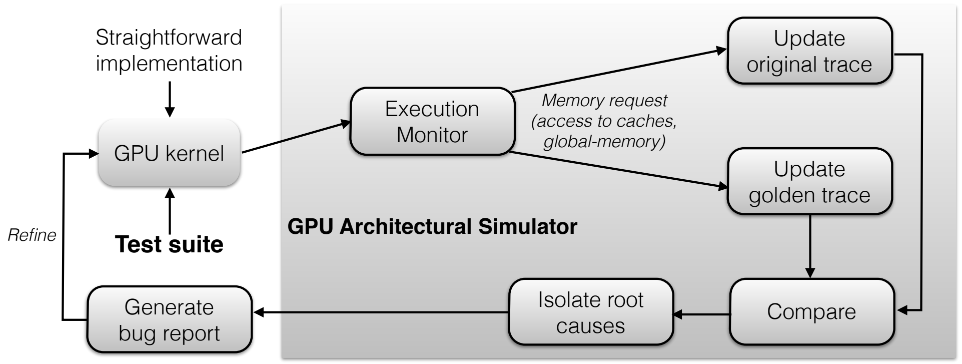


Figure 6: Overall debugging framework

as it is purely accomplished by modifying the simulator. We describe this in more detail in Section 4.1.

- Once a memory request is identified, we update the original trace and the golden trace to facilitate the debugging process. If any difference is identified between the original and the golden trace, we discover the root cause of such difference (see Sections 4.2-4.3 for details).

- After the execution is terminated, the set of root causes and the detected memory interference are used to generate a bug report. This bug report contains a prioritized list of root causes, which generate memory interference across threads. The bug report also contains suggestions for the developer to reduce such interference (see Section 4.4 for details).

4. Detailed Methodologies

Algorithm 1 outlines our overall debugging process. Our framework is carefully integrated within a simulator, such that it accurately records the memory performance of a GPGPU program. In order to accomplish this, we intercept all memory requests (*i.e.* accesses going through the cache and global-memory) during the execution and record their performance in the execution trace $\mathcal{T}$. More importantly, we manipulate the execution to compute a golden trace $\mathcal{T}'$ simultaneously. $\mathcal{T}'$ records the performance of all memory accesses *in the absence of any inter-thread interference*. It is worthwhile to mention that the computation of $\mathcal{T}'$ does not affect the original execution in $\mathcal{T}$. Besides, our framework does not require any modification of the original GPGPU program. Therefore, the execution trace

$\mathcal{T}$ accurately records the memory performance of the original GPGPU program. The main purpose of computing $\mathcal{T}'$ is to compare it with $\mathcal{T}$ and systematically locate interferences across threads. Such a comparison only requires a linear scan through the trace $\mathcal{T}$. Besides, we analyze $\mathcal{T}$ and $\mathcal{T}'$ to localize the root-cause of each inter-thread memory-interference during the execution. Finally, the detected set of inter-thread interferences and their respective root causes are used to generate a bug report. Such a bug report categorizes and prioritizes the root-cause of memory interferences. The category of a root-cause is used to suggest a possible solution to rectify memory performance bottlenecks and the respective priority is used to highlight the amount of interferences caused. In short, the generated bug report captures a concise summary to understand the memory performance problems in the respective execution scenario. In the following, we describe each component of our debugging framework in more detail.

4.1. Intercepting Memory Requests

In GPUs, several threads are often grouped together into *warps* [13] and they execute in a *lock-step* fashion. Each memory request is issued by a specific warp. However, it is possible that only a subset of threads, which constitute the warp, makes a memory request. This might happen either due to different control flow or different data usage across threads. In order to identify threads for a given memory request, a *thread mask* is provided. In our framework, we record the address of a memory block and the respective thread mask, for each memory request by a warp. Therefore, each intercepted memory request in our framework captures a tuple of the form $\langle addr, mask \rangle$, where $mask$ is a bitvector capturing the thread-mask. For instance, let us assume that a warp contains 32 threads and only the first two threads are active for a memory request at address $0xFFFF$. Such a memory request can be captured by the tuple $\langle 0xFFFF, 11\underbrace{00\ldots00}_{30} \rangle$.

A thread mask uniquely identifies the active threads *within a warp*, for a specific memory request. However, thread masks are insufficient to uniquely identify threads across warps. Since our goal is to detect memory interferences across threads, it is crucial for us to uniquely identify a thread, in a given execution. In Algorithm 1, lines 17-19 compute a unique prefix to identify a warp. In particular, a unique thread identity can be computed by understanding the execution behavior of GPGPU programs. For instance, a GPU kernel can be divided into several blocks and each block might be divided into several warps (*cf.* Figure 1). Finally, one or more warps may run on a given SIMD core. Therefore, given a thread mask $mask$, we concatenate identities of the respective block and the warp to compute a unique thread identity within a core (*cf.* lines 17-19 in Algorithm 1). Finally, the identity of the SIMD core can be used to uniquely identify threads across cores.

Algorithm 1 Localizing GPU's memory bottlenecks

```
1:  Input:
2:  $\langle \mathcal{P}, \mathcal{I} \rangle$ : $\langle$ The GPU-kernel under test, program input $\rangle$
3:  Output:
4:  $\mathcal{R}$: A bug report
5:  Launch $\mathcal{P}$ on input $I$
6:  /* initialize private cache memory for each thread */
7:  /* this is required only for debugging */
8:  if (warp $w$ is launched for the first time on core $\mathcal{C}$) then
9:      Let $N_w$ be the number of threads in $w$
10:     Let $w$ belong to block $\mathcal{B}$
11:     $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ := $empty, \forall n \in [1, N_w]$
12: end if
13: for (each memory request by a warp $w$) do
14:     Let the program location for the request be $loc$
15:     Let the memory request be $\langle addr, mask \rangle$
16:     /* construct a prefix to uniquely identify threads */
17:     Let warp $w$ belong to block $\mathcal{B}$
18:     Let warp $w$ execute on SIMD core $\mathcal{C}$
19:     Construct unique thread-identity prefix as $\mathcal{C}.\mathcal{B}.w$
20:     Let $\Omega(\mathcal{C})$ be the cache assigned to core $\mathcal{C}$
21:     /* compute the cache access category for $addr$ */
22:     $\delta$ := $\Delta(addr, \Omega(\mathcal{C}))$
23:     Update the cache $\Omega(\mathcal{C})$
24:     Say $\chi$ is the evicted memory block from $\Omega(\mathcal{C})$
25:     /* update the original execution trace */
26:     /* root causes are computed in line 55 */
27:     $\mathcal{T}$:= $\mathcal{T}$ $\bullet$ $\langle \mathcal{C}.\mathcal{B}.w, loc, addr, \delta, \chi, empty \rangle$
28:     $n$ := $1$
29:     /* manipulate the cache private to each thread */
30:     /* these caches were created for debugging at */
31:     lines 8-11 */
32:     while ($mask \neq 0$) do
33:         $\delta'[n]$ := $\varphi$
34:         if ($mask$ & $0x1$) then
35:             Construct unique thread id $\mathcal{C}.\mathcal{B}.w.n$
36:             Locate the private cache of the thread $\mathcal{C}.\mathcal{B}.w.n$
```

Algorithm 1 Localizing GPU's memory bottlenecks (continued)

```
37:             Let $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ be the private cache
38:             $\delta'[n] := \Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n))$
39:             Update the cache $\Omega(\mathcal{C}.\mathcal{B}.w.n)$
40:         **end if**
41:         $mask := mask >> 1$
42:         $n := n + 1$
43:     **end while**
44:     /* check whether any thread has a cache hit */
45:     /* in the golden execution trace $\mathcal{T}'$ */
46:     **if** $(\exists k.\ \delta'[k] \neq \varphi \wedge \delta'[k] = hit)$ **then**
47:         $\delta_g := hit$
48:     **else**
49:         $\delta_g := miss$
50:     **end if**
51:     /* update the golden execution trace */
52:     $\mathcal{T}' := \mathcal{T}' \bullet \langle \mathcal{C}.\mathcal{B}.w.k, loc, addr, \delta_g, -, - \rangle$
53:     /* find the root cause */
54:     **if** $(\delta \neq hit)$ **then**
55:         LocalizeRoot($addr, loc, \mathcal{C}, \mathcal{T}, \mathcal{T}', \delta, \delta_g, \mathcal{R}$)
56:     **end if**
57: **end for**
58: Present the bug report $\mathcal{R}$ to the developer
```

In Algorithm 1, lines 27-52 outline the manipulation of the original trace $\mathcal{T}$ and the golden trace $\mathcal{T}'$, for each memory request. An execution trace can be described as a sequence of sextuples of the form $\langle tid, pc, block, mstat, evict, root \rangle$. Each sextuple captures a memory request that traverses through the cache or global memory. The components of a sextuple have the following interpretation:

- tid : The unique identity of a warp (for the original trace $\mathcal{T}$) or a thread (for the golden trace $\mathcal{T}'$) making the memory request.
- pc : Program location invoking the memory request.
- $block$: The address of the requested memory block.
- $mstat$: Cache-access category (i.e. hit/$miss$/$miss^*$).
- $evict$: The address of the memory block evicted (or $empty$ if none evicted) from the cache due to the respective memory access.

- $root$: The root cause of any inter-thread interference faced by the memory access.

For the golden trace, we do not require the fields $evict$ and $root$. Therefore, while computing the information in the golden trace, we ignore these fields. In the following, we shall describe how the information in each sextuple is computed.

After a memory access $\langle addr, mask \rangle$ is identified, we check whether the memory request can be satisfied from the cache. Let us assume $\mathcal{C}$ is the core which makes the memory request and $\Omega(\mathcal{C})$ is the cache assigned to the core. We compute the cache-access categorization $\Delta(addr, \Omega(\mathcal{C}))$ of a memory request $\langle addr, mask \rangle$ as follows:

$$\Delta(addr, \Omega(\mathcal{C})) = \begin{cases} miss, \text{if } \epsilon(\Omega(\mathcal{C})) \wedge addr \notin \Omega(\mathcal{C}); \\ miss^{*}, \text{if } \neg\epsilon(\Omega(\mathcal{C})) \wedge addr \notin \Omega(\mathcal{C}); \\ hit, \text{otherwise}; \end{cases} \tag{1}$$

The predicate $\epsilon(\Omega(\mathcal{C}))$ is evaluated to the logical formula *false*, if and only if all cache lines in $\Omega(\mathcal{C})$ are occupied by some memory block requested by the GPGPU program. Therefore, we use the categorization $miss^{*}$ to approximate capacity misses. We also compute the evicted memory block χ from the cache $\Omega(\mathcal{C})$ (line 24 in Algorithm 1). It is worthwhile to note that an access to $addr$ may not evict any memory block from the cache. In such cases, χ gets the value of an empty block $empty$. The primary purpose of computing the evicted block χ is to identify the root cause of inter-thread interference.

We go through each thread captured by the thread mask $mask$ (lines 32-42 in Algorithm 1). We first compute a unique thread identity $\mathcal{C}.\mathcal{B}.w.n$ (line 35), using the identity of the respective core ($\mathcal{C}$), the thread block ($\mathcal{B}$), the warp (w) and the thread (n) requesting the memory. Such a unique thread identity is first used to locate the cache $\Omega(\mathcal{C}.\mathcal{B}.w.n)$, which is private to the respective thread. It is important to emphasize that the cache $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ does not really exist; it is an artifact created, manipulated and released by our framework in order to produce the golden trace and only used for the purpose of debugging. *Moreover, manipulating these caches does not affect the original execution in $\mathcal{T}$.* Once the private cache $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ is located, we compute whether the access to $addr$ can be satisfied from $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ as follows:

$$\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n)) = \begin{cases} miss, \text{if } addr \notin \Omega(\mathcal{C}.\mathcal{B}.w.n); \\ hit, \text{otherwise}; \end{cases} \tag{2}$$

Since the cache $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ is private to the thread $\mathcal{C}.\mathcal{B}.w.n$, $\Omega(\mathcal{C}.\mathcal{B}.w.n)$ does not suffer any inter-thread interference. As a result, $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n))$ can be compared with $\Delta(addr, \Omega(\mathcal{C}))$ to discover the effect of inter-thread memory interferences. However, we note that each memory request in the golden execution might be evaluated differently for different threads within the same warp. Specifically, if both threads n and n' belong to the same warp w, $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n))$ might not be equal to $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n'))$. In such cases, we consider two different scenarios as follows:

- $\mathcal{S}1$: $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n)) = hit$ for at least one thread n within a warp,
- $\mathcal{S}2$: $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n)) = miss$ for all threads n within the same warp,

Let us assume that $\Delta(addr, \Omega(\mathcal{C}))$ was not evaluated to be a hit. For the scenario $\mathcal{S}1$, we can argue that a thread n, for which $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n)) = hit$, suffers a cache miss in the original execution due to interference across threads. For $\mathcal{S}2$, all threads within the same warp suffer cache misses, even in the absence of interference across threads. Therefore, we consider that the interference across threads was not the cause of the resulting cache miss. In Algorithm 1, Scenarios $\mathcal{S}1$ and $\mathcal{S}2$ are considered in lines 44-49.

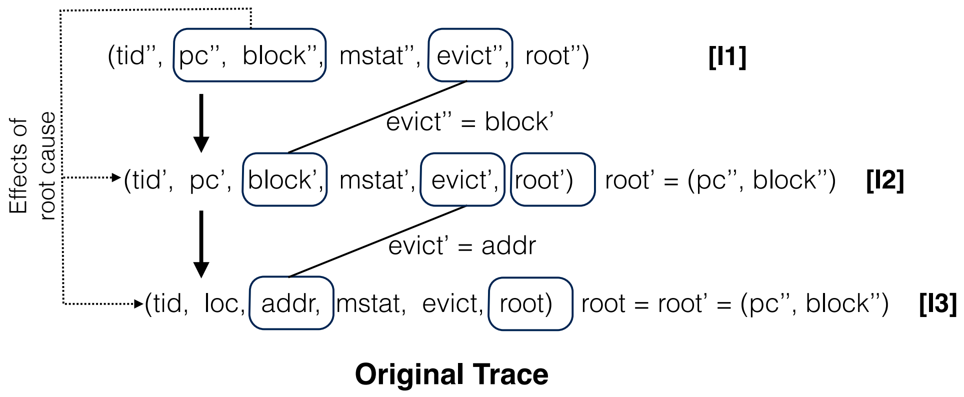
We update the execution traces $\mathcal{T}$ (the original trace) and $\mathcal{T}'$ (the golden trace) with the information computed via $\Delta(addr, \Omega(\mathcal{C}))$ and $\Delta(addr, \Omega(\mathcal{C}.\mathcal{B}.w.n))$, respectively. More importantly, we use this information to compute the root cause of memory interferences. This is accomplished via the procedure `LocalizeRoot`, described in Section 4.3.

Finally, we continue the aforementioned process for each memory request by a warp (lines 13-55 in Algorithm 1).

4.3. Locating Root Causes

Figure 7 outlines the procedure to localize the root cause of memory performance bottlenecks. In Algorithm 1, this procedure is carried out via line 55 (call to procedure `LocalizeRoot`). The key idea is to check the difference between the original and the golden trace and to traverse the chain of memory blocks evicted from the cache. The memory access, of the first eviction in such a chain of memory blocks, is determined to be a root cause of memory interference.

Let us consider our example in Figure 2 and assume that we want to find the root cause for the memory access captured by $\langle m1, miss, m2\rangle$. We go backwards in the trace and observe that $m1$ has been evicted by $m1'$. Therefore, the root cause of interference at $\langle m1, miss, m2\rangle$ was $m1'$. However, for the memory access $\langle m2, miss, m1'\rangle$, although $m2$ has been evicted by $m1$, we highlight the root cause



(tid'', pc'', block'', mstat'', evict'', root'')
[I1]
evict'' = block'
(tid', pc', block', mstat', evict', root')
root' = (pc'', block'')
[I2]
evict' = addr
(tid, loc, addr, mstat, evict, root)
root = root' = (pc'', block'')
[I3]
Effects of root cause
Original Trace

Figure 7: Showing root-cause finding for a sequence of three instructions $I1, I2$ and $I3$, assuming all instructions suffer cache misses. If $block''$ has not been accessed before, $I1$ will *inevitably* suffer a cache miss and we make its root cause to be *empty*. If $I1$ evicts $block'$ (*i.e.* $block' = evict''$), we set $root' = (pc'', block'')$ while executing $I2$. If $I2$ evicts $addr$ (*i.e.* $addr = evict'$), we have $root$ to be the same as $root'$ (*i.e.* the root cause affecting $I2$). Therefore, in this example, there is a single root cause $(pc'', block'')$ that results in cache misses both at $I2$ and $I3$.

to be $m1'$. This is because, the root cause for the eviction of $m1$ was $m1'$, as discovered on-the-fly.

For each memory request, we first compare cache-access categories δ and δ_g, which are passed as parameters to the procedure call `LocalizeRoot` (*cf.* line 55 in Algorithm 1). Note that δ was derived from the original trace, whereas δ_g was derived from the golden trace. Therefore, if δ was observed to be a cache hit, then the respective memory request does not affect performance. For the rest of the scenarios, we categorize memory interferences into different types of faults as shown in Table 1 (note that $\mathcal{R}$ captures the bug report).

Fault type	Scenario
$\mathcal{R}.fault_{mh}$	$(\delta = miss \wedge \delta_g = hit)$
$\mathcal{R}.fault_{m^*h}$	$(\delta = miss^* \wedge \delta_g = hit)$
$\mathcal{R}.fault_{mm}$	$(\delta \neq hit \wedge \delta_g = miss)$

Table 1: Categorizing fault types

We note the special case when both the original and the golden trace observe cache misses (*i.e.* Fault $\mathcal{R}.fault_{mm}$). For such cases, we can conclude that inter-thread interferences do not cause the cache misses. Therefore, reducing the interference among threads will not result in an improved memory performance. It is possible that cache misses were unavoidable for both the original and the golden trace (*e.g.* first few accesses in Figure 2 - Figure 5). Otherwise, the cache misses were caused by memory interferences within a thread. One possible way to reduce such cache misses is to reduce the global-memory usage in each thread. This, in turn, will potentially reduce the memory interference caused within a thread.

We formulate a fault type f (*i.e.* $\mathcal{R}.fault_{mh}$, $\mathcal{R}.fault_{m^*h}$ or $\mathcal{R}.fault_{mm}$) by a tuple $\langle count, causes \rangle$, where $count$ captures the number of times f appears in the execution and $causes$ captures all root causes that result in fault type f. Each root cause in a fault type is further decomposed into its priority p and all possible locations $effects$, which are affected by the respective root cause. We compute the priority (p) of a root cause via the number of interferences caused by it. Each entry in $effects$ is captured by a tuple $\langle loc, addr \rangle$, where loc is the program location and $addr$ is the accessed memory block. For instance, in Figure 7, we have a single root cause causing two faults. Assuming both faults belong to the same fault-type, we record the following information: $\langle pc'', block'' \rangle \in \mathcal{R}.causes$, $\langle pc', block' \rangle \in \mathcal{R}.causes[\langle pc'', block'' \rangle].effects$, and $\langle loc, addr \rangle \in \mathcal{R}.causes[\langle pc'', block'' \rangle].effects$. Figure 8 generalizes the structure of a fault type in the bug report.

4.4. Generating Bug Report

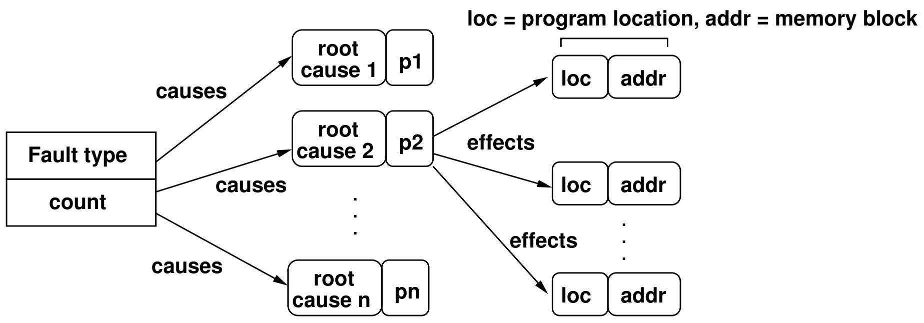


Figure 8: Structure of the bug report for each fault type

The generated bug report (*cf.* line 58 in Algorithm 1) summarizes the memory-performance bottlenecks detected during an execution. Figure 8 graphically captures the structure of a bug report for a given fault type. The $count$ subfield captures the number of occurrences of the respective type of fault. Therefore, depending on the value of the $count$ subfield, the developer can prioritize the order of fixing a fault. We emphasize that the type of the fault allows a developer to find its possible fix. In particular, Table 2 distinguishes different faults according to their potential solutions.

Fault type	**Potential fix**
$\mathcal{R}.fault_{mh}$	Improving cache management (better cache mapping, changing data-access pattern)
$\mathcal{R}.fault_{m^*h}$	Reducing number of threads or reducing accesses to the cache via shared memory
$\mathcal{R}.fault_{mm}$	Reducing global-memory usage within thread (*e.g.* using auxiliary registers)

Table 2: Categorizing fault types according to their fixes

To summarize, we generate a bug report $\mathcal{R}$, which categorizes inter-thread interferences into three different categories (*cf.* Table 2). For each such category, we list the set of root causes (*cf.* Section 4.3). Each root cause is prioritized (as captured via $p1, p2, \ldots, pn$ in Figure 8) according to the number of inter-thread interferences caused by the same. Therefore, the priority of a root cause can be used to rectify the most significant interferences early during the design process. Finally, our bug report $\mathcal{R}$ satisfies the following properties:

Property 4.1. *For a given execution scenario, let us assume f is a fault (cache miss) of type $\mathcal{R}.fault_{mh}$ or of type $\mathcal{R}.fault_{m^*h}$. This fault appears due to the interferences between threads.*

Property 4.2. *For a given execution, if the interference among threads leads to a cache miss of memory block "$addr$" at program location "loc", then the bug report $\mathcal{R}$ must include a root cause r and a fault type f, such that $\langle loc, addr\rangle \in \mathcal{R}.causes[r].effects$.*

5. Evaluation

Experimental setup. We have implemented GRAB [12] on top of GPGPU-Sim [11]. We use the `nvcc` compiler to compile GPU kernels into CUDA compliant binaries, execute them using GPGPU-Sim and manipulate the execution via GRAB. Our implementation is completely *automated* and it does not require any manual intervention. Besides, we do not need to modify the underlying GPGPU program to place it into the debugging environment. Since we aim to uncover memory bottlenecks, we choose GPU kernels with substantial data accesses. All our subject programs are available in [12]. Table 3 reports some salient features of the subject

Program name	Input size (bytes)	#Threads/block	#Blocks
`Transpose`	1024 * 256 * 4	256	1024
`Bitonic_Sort`	512 * 4	512	256
`Scan`	512 * 4	512	1
`MatMult`	64 * 64 * 4	64	64
`LBM`	128 * 128 * 17 * 4	512	32
`Susan`	512 * 512	128	2048

Table 3: Salient features of the subject programs

programs (chosen from [15]) used in our evaluation. We have also chosen two case studies. The first case study implements a fluid dynamic simulation using Lattice Boltzmann Models (*cf.* `LBM` in Table 3) [16] and the second case study captures an image processing algorithm for noise filtering, edge finding and corner finding (*cf.* `Susan` in Table 3) [17]. Both `LBM` and `Susan` are suitable for GPU implementation, as their respective computational units (an image pixel for `Susan` and a node for `LBM`) could be processed in parallel by independent GPU threads. All our evaluations have been conducted on an NVIDIA Tegra K1 platform. For measuring the execution time on the Tegra K1, we have used the default frequencies: 72 MHz for the GPU's core clock and 204 MHz for the GPU's memory clock. Finally, we compute the percentage improvement using the formula $\frac{t_{old}-t_{new}}{t_{old}}$, where t_{old} is the time taken by the initial version and t_{new} is the time taken by the version obtained after improvement.

We evaluate and study the following research questions:

- ***RQ1:*** *How effective is GRAB to localize performance bottlenecks?* In particular, we aim to investigate whether memory-related bottlenecks, as detected by GRAB, play a major role in the efficiency of a GPGPU program. Besides, we would also like to see the ratio between the number of reported faults and the number of detected root causes.

- ***RQ2:*** *Can we use GRAB to improve the performance of GPGPU programs?*

- ***RQ3:*** *Can we use GRAB to select appropriate GPU platform?* In particular, we would like to discover whether GRAB can be used for hardware/software co-design, with a specific focus on GPU platforms.

RQ1: How effective is GRAB to localize performance bottlenecks?. Table 4 summarizes our evaluation. Subject programs in Table 3 have different implemen-

Program name	Execution time on Tegra K1 (μs)	$\mathcal{R}.fault_{mh}.count$			$\mathcal{R}.fault_{m^*h}.count$			$\mathcal{R}.fault_{mm}.count$		
		#Faults	#Root cause	#Aff	#Faults	#Root cause	#Aff	#Faults	#Root cause	#Aff
`Transpose_naive`	2644	0	0	0	0	0	0	0	0	0
`Transpose_best`	1750	0	0	0	0	0	0	10	1	1
`Bitonic_Sort_basic`	503	1	1	1	1898	2	6	39	2	3
`Bitonic_Sort_best`	468	0	0	0	8	1	1	0	0	0
`Scan_naive`	3870	0	0	0	0	0	0	3594	1	1
`Scan_workefficient`	4514	0	0	0	3156	1	1	3992	1	2
`Scan_best`	3373	0	0	0	0	0	0	2400	1	2
`MatMult_basic`	2184	210	3	2	6540	3	2	10039	3	3
`MatMult_basic_modified`	1482	894	2	1	5688	2	1	1792	2	2
`MatMult_best`	778	0	0	0	64	1	1	504	3	1
`LBM_basic`	6322	0	0	0	46258	15	18	49036	14	20
`LBM_V1`	4780	0	0	0	2519	9	7	13684	12	9
`LBM_V2`	4212	18	1	1	2282	5	3	4088	6	4
`Susan_basic`	6832	0	0	0	26095	17	14	12938	17	17
`Susan_modified`	6410	0	0	0	1886	3	7	1609	6	8

Table 4: Summary of evaluation (source codes of program versions are in the project website [12]). #Aff captures the number of program locations reporting faults (*affected locations*) and #Faults captures the total number of reported faults

tations to compare the GPU performance. For `Transpose` and `Scan`, all the implementations were chosen from [15]. For `Bitonic_Sort`, `MatMult` (matrix multiplication), `LBM` and `Susan` we obtained straightforward GPU implementations (*i.e.* `Bitonic_Sort_basic`, `MatMult_basic`, `LBM_basic` and `Susan_basic`, respectively). Such implementations were written by an experienced developer without the knowledge of memory subsystems. From Table 4, we make the following observations:

(OBS1). We want to compare different GPU implementations of the same program. As observed from Table 4, none of the implementations for `Transpose` report a substantial number of faults. This result guides the developer to focus *not* on reducing the interference across threads, instead, focusing on other memory-related bottlenecks (*e.g. uncoalesced memory accesses*). On the contrary, the best implementation of `Scan` (*i.e.* `Scan_best`) exhibits much less faults compared to the other two implementations (*i.e.* `Scan_naive` and `Scan_workefficient`). We make similar observations for `Bitonic_Sort`, `MatMult`, `LBM` and `Susan`. Therefore, we can conclude that the number of faults, as reported by GRAB, plays an important factor in GPU performance.

(OBS2). The number of root causes are substantially smaller than the number of reported faults. For instance, let us consider the kernel `Bitonic_Sort_basic`. Executing this kernel, we report 1898 faults of type $\mathcal{R}.fault_{m^*h}$. However, all the problems are caused by only two locations. This makes the bug report useful for investigation, as we can concentrate on a few small number of root causes. The number of affected locations is also substantially smaller than the number of faults. This is expected, as a single location in the GPGPU code might be accessed by thousands of threads. In particular, this makes the developer aware that a single location in the GPGPU code might cause a substantial number of faults, but, modifying only a few locations might significantly improve the performance.

RQ2: Can we use GRAB to improve the performance of GPGPU programs?. In order to show the usage of GRAB, we choose the straightforward implementation of `Bitonic_Sort` (*i.e.* `Bitonic_Sort_basic`). Our goal is to see whether the memory performance of `Bitonic_Sort_basic` can be improved using GRAB. We use GRAB to locate all the memory performance bottlenecks and generate the bug report (*cf.* Table 4). As observed from Table 4, GRAB reports 1898 faults of type $\mathcal{R}.fault_{m^*h}$ and 39 faults of type $\mathcal{R}.fault_{mh}$. Since the number of type $\mathcal{R}.fault_{m^*h}$ faults is substantially higher, we concentrate whether type $\mathcal{R}.fault_{m^*h}$ faults could be fixed first. Recall that type $\mathcal{R}.fault_{m^*h}$ faults might be reduced by decreasing the number of threads or by moving data into the shared memory, and thereby reducing the number of accesses to the cache (*cf.* Table 2). Fortunately, our bug report only records two root causes and six locations to be affected by type $\mathcal{R}.fault_{m^*h}$ fault. As a result, we can concentrate only on a small set of locations in the program `Bitonic_Sort_basic`. From the bug report, we can identify that these locations correspond to the access of the input array. Therefore, a potential solution is to access the input array in such a fashion that they bypass the cache. This can be accomplished by moving the input array into shared memory and modifying the code accordingly. This results in the implementation of `Bitonic_Sort_best` (similar to the one from [15]), which substantially reduces the number of faults reported by GRAB (*cf.* Table 4) and *improves the overall performance by 7% on real hardware*.

We investigated `MatMult_basic` to improve its performance using GRAB. We first aim to reduce $\mathcal{R}.fault_{mm}$ fault-type, as it has the largest number of occurrences. To rectify faults of type $\mathcal{R}.fault_{mm}$, note that we can potentially reduce the data usage in a thread (*cf.* Table 2). In order to do this, we located one of the root causes and observed that the multiplication result is repeatedly being written to the global-memory. To fix this, we simply introduced a scalar variable `sum` (which is allocated in a register), where we store the intermediate result and write to the global-memory at the end of the kernel. Note that it may potentially increase the code size, which is the reason, the developer might have avoided doing so in the first place. Our fix eliminates all reported faults of `MatMult_basic` that were generated by the respective root cause. Table 4 captures this result via `MatMult_basic_modified`. This fix improves the overall performance by 32%. Subsequently, we focus on eliminating faults of type $\mathcal{R}.fault_{m^*h}$, as it occurs 5688 times in the bug report. GRAB pinpoints that accesses to both input matrices are exhibiting the faults. A potential fix for fault-type $\mathcal{R}.fault_{m^*h}$ is to move data into shared memory (*cf.* Table 2). Therefore, we modify the implementation such that input matrices are accessed from shared memory. Due to the limited amount of shared memory, such accesses to shared memory must be performed in separate chunks. These modifications lead to the solution `MatMult_best` (similar to the one from [15]), which dramatically reduces the number of reported faults (*cf.* Table 4) and *improves the overall performance by 64% on real hardware*.

For `LBM_basic`, we focused on improving only one of the kernels (called `periodic_boundary` in [12]). Table 4 reports that fault-type $\mathcal{R}.fault_{mm}$ has the largest number of occurrences. This fault-type had multiple root causes. In order to improve the performance, we focused on the root cause that generated the largest number of faults (22574 in our evaluation). We observed that this root cause indicates reading a global-memory location within a loop (*cf.* Figure 9(a)). To fix this, we loaded the value in a register and replaced all its use via the register within the loop body. This transformation is shown in Figure 9(b). However, we observed that the instruction ``reg aux1 = glob j'' (*cf.* Figure 9(b)) induces 21140 faults of type $\mathcal{R}.fault_{mm}$. In order to reduce this effect, we analyzed the affected locations and observed that the global-memory usage can further be reduced using the modification in Figure 9(c). In particular, this modification reuses the same global-memory location via the register `aux2`.

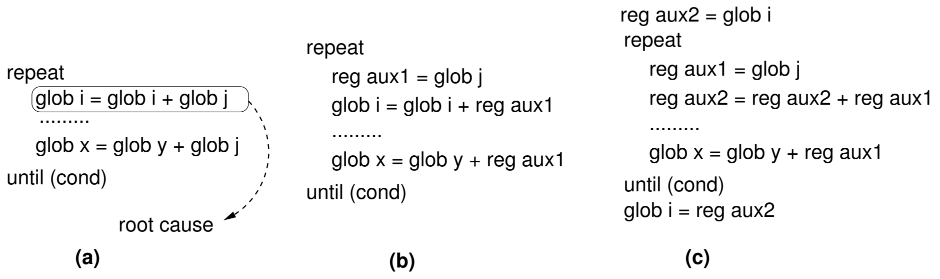


Figure 9: Modification strategies for Lattice Boltzmann case study. Keyword `glob` indicates the usage of global-memory, whereas the keyword `reg` indicates the usage of auxiliary registers

Due to the nature of computations in the loop, we observed that modifications, as in Figure 9, can be employed to two more statements that repeatedly read from global-memory. This leads to `LBM_V1` that improves the performance *by 24%* (*cf.* Table 4). `LBM_V1` still had 13684 faults of type $\mathcal{R}.fault_{mm}$, where 9063 were caused by a single root cause. This root cause was located in the second loop of the kernel. We could apply the same strategy as in Figure 9. However, we also noted that data-usage in the first-half of the second loop overlapped with the data-usage in the first loop. We, therefore, split the second loop in half and fused the top-half with the first loop. This modification leads to `LBM_V2`, which has significantly less reported faults of type $\mathcal{R}.fault_{mm}$ and *improves the overall performance by 33%*. We also compared the implementation `LBM_V2`, as obtained by using hints from GRAB and an implementation created by an experienced GPU programmer (available in [12]) without using GRAB. *Our comparison revealed that the implementation `LBM_V2` outperforms the implementation generated by the GPU programmer.*

For `Susan_basic`, `susan_principle_small_gpu` [12] is the kernel we focused on improving. Table 4 reports that fault-type $\mathcal{R}.fault_{m*h}$ has the largest number of occurrences. As specified in Table 2, such faults can be reduced by placing data into shared memory. Using the generated bug report, we located the data that exhibit fault of type $\mathcal{R}.fault_{m*h}$. Then we modified the code to initially load the respective data from global memory into shared memory and subsequently, perform all the computations in shared memory. We did this by restructuring the GPU kernel to use as many threads in a thread block as the image width (512 in our case, see Table 3). This produced an implementation `Susan_modified`, which improves performance *by 6%*. It is worthwhile to mention that increasing the number of threads per block in `Susan_basic` to 512 did not change the execution time.

Figure 10 shows the improvement percentage of the modified code versions, compared to the original ones. The data has been compiled from the column “Execution time” (*i.e.* second column) in Table 4.

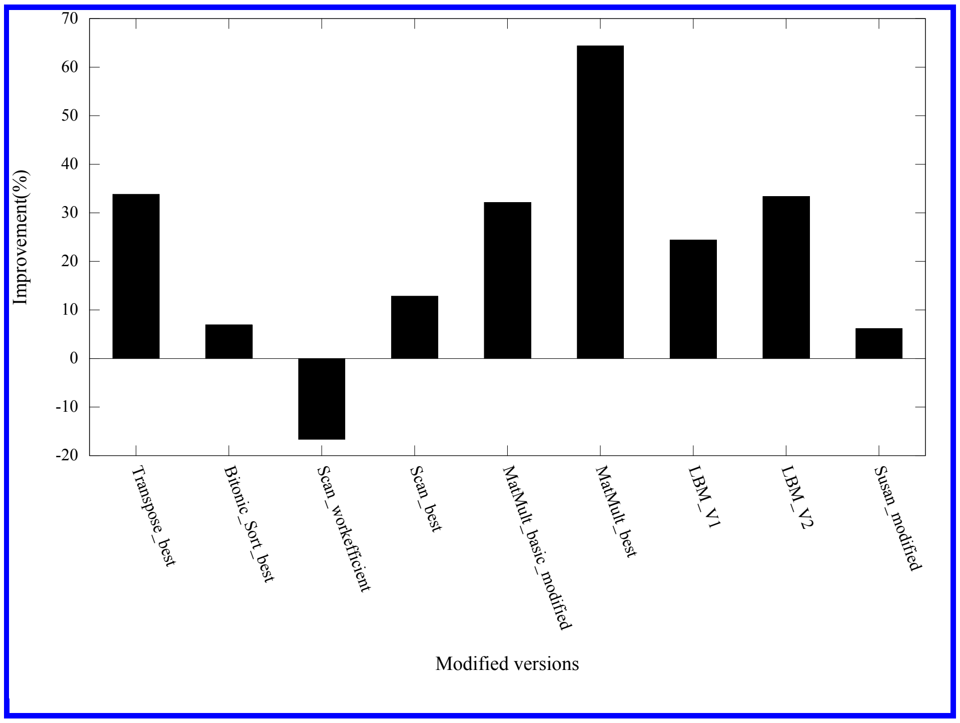


Figure 10: Relative improvement in execution time with respect to the original versions

In order to evaluate whether the change guided via GRAB could lead to improvement in general, we have run the modified implementations with 100 random input data sets. Figure 11 captures the average, minimum and maximum improvements over such randomly generated inputs. As observed from Figure 11, only `Bitonic_Sort_Best` does not provide better average performance, as compared to the original implementation. As generated reports of GRAB are based on test executions, in general, the suggestion of GRAB depends on the quality of tests. For instance, GRAB-generated reports may not lead to improvement for a new test, if such a test explores code not covered by the test suite used by GRAB. Due to this limitation, we observe negative improvements in average performance for `Bitonic_Sort`. However, we believe that generation of better test cases, in order to expose memory-related bottlenecks, is orthogonal to the problem addressed in the paper. Of course, GRAB can always be combined with sophisticated test generation strategies that are specifically tailored to find memory-related bottlenecks as well as obtaining better code coverage.

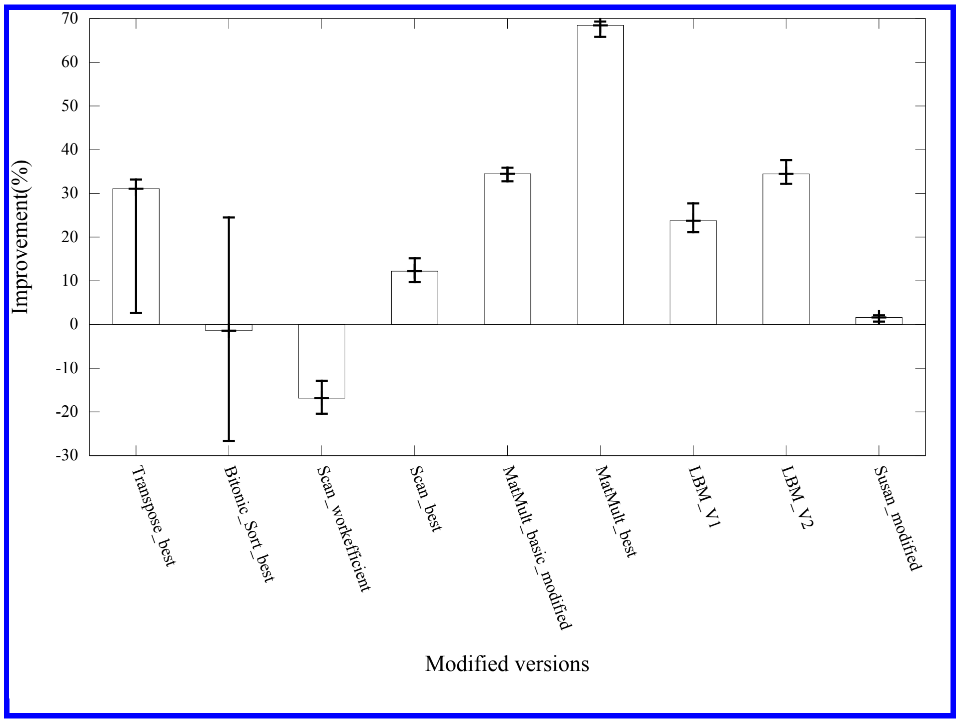


Figure 11: Variation of improvement for random input data sets

Preceding examples demonstrate how GRAB can systematically help to improve basic implementations to more efficient versions, either manually or via hints to guide a compiler. Table 4 clearly indicates that the guidance obtained from GRAB can generate efficient GPGPU programs, leading to improvements of up to 64%, on real hardware.

RQ3: Can we use GRAB to select appropriate GPU platform?. We also evaluate our debugging framework with different cache configurations as seen in Figure 12. Furthermore, Figure 12 is useful when the execution platform is not available and the system designer wants to select an appropriate GPU platform. For instance, consider the Vivante GC2000 embedded GPU, which does not include a shared memory, but includes a 4KB cache. From Figure 12, we can conclude that `Scan_best`, `Scan_naive` and `Bitonic_best` will run efficiently on this GPU, as the number of interferences drop to zero with a 2KB cache. However, the implementation `Bitonic_Sort_basic` will not run efficiently in the Vivante GPU. In the preceding paragraph, we observed that an efficient version of `Bitonic_Sort` (*i.e.* `Bitonic_Sort_best`) can potentially use shared memory to reduce the interference in the cache. Since the Vivante GPU does not feature a shared memory, the designer may potentially like to choose a different GPU platform for the `Bitonic_Sort` kernel.

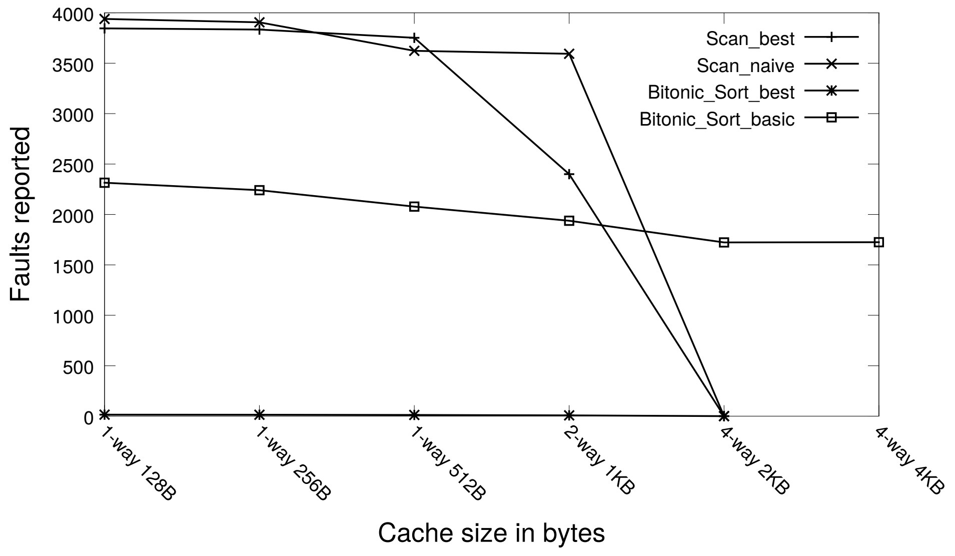


Figure 12: Faults reported with respect to different cache size

Scheduling policies. It is worthwhile to mention that the GPU performance may vary depending on the actual thread-scheduling policy. In order to stress this point, we ran both basic and improved versions of GPU kernels for different scheduling policies. Note that such an experiment can only be performed in a simulator, as we cannot control the scheduling policy in real hardware. Table 5 reports our findings. Table 5 clearly indicates that although the actual improvement might vary, the fixes suggested by GRAB leads to more efficient GPU kernels regardless of the scheduling policy.

Program name	Simulation cycles		
	Greater then oldest (GTO)	Loose round robin (LRR)	Two level (TL)
`Bitonic_Sort_basic`	31837	44265	36009
`Bitonic_Sort_best`	16446	15886	19250
`MatMult_Basic`	118434	112738	112494
`MatMult_Basic_modified`	110023	105664	106682
`MatMult_best`	22831	20604	21872
`LBM_basic`	470074	500903	494320
`LBM_V1`	254566	266084	248478
`LBM_V2`	176690	187940	174754
`Susan_basic`	176779	145125	167513
`Susan_modified`	53701	48891	65189

Table 5: Evaluation for different scheduling policies

Debugging time. Our experiments were performed on a machine having a 2.9 GHz Intel Core i5-4210H CPU and 12 GB RAM, running Ubuntu 14.04, 64-bit operating system. The overhead of GRAB is negligible and is only about **5%** of the total execution time of the simulator. The maximum execution time of the simulator, in turn, was four minutes, as observed for `LBM_Basic`. The negligible overhead appears due to the efficient comparison between the original trace and the golden trace (linear with respect to the length of execution). Moreover, we avoid repeatedly searching the same root cause (*cf.* Figure 7) to reduce the debugging overhead.

6. Related work

Profiling. Understanding the performance behavior of programs is a critically important problem. The *state-of-the-art* in performance debugging has long been *profiling* [4, 5, 18, 19, 20]. Profilers are inadequate to fix performance bugs, due to their inability to precisely highlight the location of *performance wastage* (instead of hotspots) and root causes resulting such wastage. Although a recent work [21] has proposed to highlight potential scalability problems in MPSoC platforms, it is unable to provide the cause of such problems. In this paper, we aim to highlight the cause of performance problems in the memory subsystems (*e.g.* caches), with a specific focus on GPGPU programs. Moreover, our framework guarantees to discover all cache misses that occur due to interferences across threads.

Record and replay. Replay debuggers [22, 23, 24, 25] can be used to record the execution order between threads during a production run and the same order is faithfully replayed to reproduce production-run failures. In contrast, our approach automatically generates a hypothetical golden execution from the original execution. This golden execution is further used to localize memory interferences in GPGPU programs and more importantly, to localize the cause of such interferences.

Automated debugging. Works on automated debugging [26, 27] have made major inroads in the past few decades. Nevertheless, most of these works have primarily concentrated on debugging functionality-related bugs. In general, debugging performance-related bugs are more challenging than debugging functionality-related bugs. This is due to the fact that performance bugs critically depend on the underlying execution platform and there is a lack of understanding to clearly distinguish a buggy execution in terms of performance. In this paper, we concentrate primarily on GPUs as the execution platform and we derived an automated approach to distinguish execution scenarios that exhibit performance bugs.

Fault localization. Automated techiques on fault localization [28] aim to discover the location of faults or the root cause of a failure. Automatic localization of functionality-related faults is still an active research area. To find the root cause of a performance-related fault, we face some unique challenges due to the lack of appropriate timing models in programs. Therefore, existing techniques on automated fault localization cannot be directly adopted for localizing performance-related faults. In this paper, we provided a novel approach to compare a failed execution (in terms of performance) with an automatically computed golden execution. Moreover, we also provided a methodology to leverage the information from such executions and compute the root cause of memory-related failures in GPUs.

Performance testing. Recent works on detecting performance bottlenecks [29, 30] primarily concentrate at the software level, such as redundant computations and misusage of function calls. On the contrary, we argue the importance of execution platforms in the context of detecting performance bugs, with a specific focus on GPUs. Our previous works on performance testing [14, 31] or other works on testing GPGPU programs [32] concentrate on test-input generation and are not directly applicable for localizing the root cause of performance bugs. In this paper, we aim to localize the cause of memory interferences by systematically comparing the original and the golden traces.

Empirical performance model. The performance characteristics of GPUs have recently been studied via analytical models [33, 34, 35, 36]. In contrast to these works, our approach has a significant flavor of software debugging, as we aim to systematically highlight the root cause of memory-performance bottlenecks. Besides, our approach does not depend on the approximation incurred in analytical models.

Worst-case execution time analysis. In recent years, works on worst-case execution time (WCET) analysis for GPGPU programs has gained attention [37, 38]. Our approach is orthogonal to the approach taken in WCET analysis. Our aim is not to statically predict a bound on WCET, instead we aim to concentrate on a more general notion of memory performance bugs. In other words, our methodology has a significant flavor in terms of testing and debugging, compared to the approaches proposed for WCET analysis. Besides, as our approach analyzes concrete executions, it does not suffer from the imprecision incurred in static WCET analysis.

In summary, we extend the foundation of automated debugging and fault localization via localizing the cause of memory-related bottlenecks, with a specific focus on GPGPU programs.

7. Discussion

In this paper, we have proposed GRAB, a systematic framework to localize memory performance bottlenecks in GPGPU programs. We demonstrate the usage of GRAB via several experiments on an NVIDIA Tegra K1 platform. Although GRAB uses a simulator, we show that it is not needed for the simulator to implement exactly the same scheduler as the target platform, which is impossible since the actual scheduling policies are not known. This has been demonstrated practically by running the programs on the real hardware and the produced performance improvement (*cf.* Table 4). Besides, we have also shown that the fixes suggested by GRAB improves performance regardless of the scheduling policy (*cf.* Table 5).

It is worthwhile to mention that we only detect performance bottlenecks due to the interference across threads. GPGPU programs may also suffer from other performance issues, such as due to uncoalesced memory accesses or due to an imbalanced workloads among threads. Detection of uncoalesced memory accesses can be reported by existing NVIDIA profilers [8]. However, the detection of imbalanced workloads requires deeper analysis of GPGPU programs. GRAB is only a first step towards building a performance debugger for GPGPU programs and we are working to extend GRAB to detect more advanced performance bugs, such as imbalanced workloads.

References

[1] NVIDIA Tesla, http://www.nvidia.com/object/tesla-supercomputing-solutions.html.

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

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

[4] T. Ball, J. R. Larus, Efficient path profiling, in: MICRO, 1996.

[5] J. R. Larus, Whole program paths, in: PLDI, 1999.

[6] Collect performance data with the collector, http://docs.oracle.com/cd/E18659_01/html/821-2763/gkofq.html#scrolltoc.

[7] Valgrind instrumentation framework, http://valgrind.org.

[8] NVIDIA GPU Profiler, https://developer.nvidia.com/nvidia-visual-profiler.

[9] ARM Streamline Performance Analyzer, http://ds.arm.com/ds-5/optimize/.

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

[11] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, T. M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: ISPASS, 2009.

[12] GRAB: systematic detection of memory related performance bottlenecks in GPGPU programs, http://www.ida.liu.se/~adrho74/project/grab.shtml.

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

[14] S. Chattopadhyay, P. Eles, Z. Peng, Automated software testing of memory performance in embedded gpus, in: EMSOFT, 2014.

[15] CUDA SDK code samples, `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/samples.html`.

[16] D. T. Thorne, C. Michael, Lattice Boltzmann modeling: An introduction for geoscientists and engineers, Springer, 2006.

[17] S. M. Smith, J. M. Brady, SUSAN A new approach to low level image processing, International journal of computer vision 23 (1).

[18] E. Coppa, C. Demetrescu, I. Finocchi, Input-sensitive profiling, in: PLDI, 2012.

[19] D. Zaparanuks, M. Hauswirth, Algorithmic profiling, in: PLDI, 2012.

[20] M. Kim, P. Kumar, H. Kim, B. Brett, Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model, in: IPDPS, 2012.

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

[22] G. Altekar, I. Stoica, ODR: output-deterministic replay for multicore debugging, in: SOSP, 2009.

[23] D. Weeratunge, X. Zhang, S. Jagannathan, Analyzing multicore dumps to facilitate concurrency bug reproduction, in: ASPLOS, 2010.

[24] D. Hower, M. D. Hill, Rerun: Exploiting episodes for lightweight memory race recording, in: ISCA, 2008.

[25] J. Huang, C. Zhang, J. Dolby, CLAP: recording local executions to reproduce concurrency failures, in: PLDI, 2013.

[26] A. Zeller, Isolating cause-effect chains from computer programs, in: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, ACM, 2002, pp. 1–10.

[27] H. Agrawal, R. A. DeMillo, E. H. Spafford, Debugging with dynamic slicing and backtracking, Software: Practice and Experience 23 (6) (1993) 589–616.

[28] J. A. Jones, M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 2005, pp. 273–282.

[29] G. Jin, L. Song, X. Shi, J. Scherpelz, S. Lu, Understanding and detecting real-world performance bugs, in: PLDI, 2012.

[30] A. Nistor, L. Song, D. Marinov, S. Lu, Toddler: detecting performance problems via similar memory-access patterns, in: ICSE, 2013.

[31] A. Banerjee, S. Chattopadhyay, A. Roychoudhury, Static analysis driven cache performance testing, in: RTSS, 2013.

[32] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. P. Rajan, GKLEE: Concolic verification and test generation for GPUs, in: PPoPP, 2012, `http://www.cs.utah.edu/formal_verification/GKLEE/`.

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

[34] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, W.-m. W. Hwu, An adaptive performance modeling tool for GPU architectures, in: PPoPP, 2010.

[35] J. Sim, A. Dasgupta, H. Kim, R. W. Vuduc, A performance analysis framework for identifying potential benefits in GPGPU applications, in: PPOPP, 2012.

[36] J. H. Lee, J. Meng, H. Kim, SESH framework: A space exploration framework for GPU application and hardware codesign, in: PMBS, 2013.

[37] A. Betts, 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.

[38] K. Berezovskyi, L. Santinelli, K. Bletsas, E. Tovar, WCET measurement-based and extreme value theory characterisation of cuda kernels, in: Proceedings of the 22nd International Conference on Real-Time Networks and Systems, ACM, 2014, p. 279.