Efficient Exception Handling Support for GPUs

Ivan Tanasic[1,2], Isaac Gelado[3], Marc Jorda[1], Eduard Ayguade[1,2], Nacho Navarro[1,2]
[1]Barcelona Supercomputing Center, [2]Universitat Politecnica de Catalunya, [3]NVIDIA
{itanasic, mjorda, eduard, nacho}@ac.upc.edu, igelado@nvidia.com

ABSTRACT

Operating systems have long relied on the exception handling mechanism to implement numerous virtual memory features and optimizations. However, today's GPUs have a limited support for exceptions, which prevents implementation of such techniques. The existing solution forwards GPU memory faults to the CPU while the faulting instruction is stalled in the GPU pipeline. This approach prevents preemption of the faulting threads, and results in underutilized hardware resources while the page fault is being resolved by the CPU.

In this paper, we present three schemes for supporting GPU exceptions that allow the system software to preempt and restart the execution of the faulting code. There is a trade-off between the performance overhead introduced by adding exception support and the additional complexity. Our solutions range from 90% of the baseline performance with no area overheads, to 99.2% of the baseline performance with less than 1% area and 2% power overheads. Experimental results also show 10% performance improvement on some benchmarks when using this support to context switch the GPU during page migrations, to hide their latency. We further observe up to 1.75x average speedup when implementing lazy memory allocation on the GPU, also possible thanks to our exception handling support.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; *Parallel architectures*; • **Software and its engineering** → Virtual memory;

KEYWORDS

Exceptions, page fault, virtual memory, context switch, GPU.

ACM Reference format:
Ivan Tanasic[1,2], Isaac Gelado[3], Marc Jorda[1], Eduard Ayguade[1,2], Nacho Navarro[1,2]. 2017. Efficient Exception Handling Support for GPUs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123950

1 INTRODUCTION

GPU accelerated systems have traditionally put the programmer in charge of keeping the CPU and GPU memories coherent using explicit DMA transfers to copy the data between them [35]. Recently, AMD and NVIDIA have commercialized discrete GPUs with support for automatic data transfers between CPU and GPU memories [26, 36]. Memory coherence is achieved through a software implementation of the DSM (*Distributed Shared Memory*) model that exploits page faults to perform page migrations between the CPU and GPU memories [58]. On-demand page migration finally removes the need for explicit data transfers, drastically improving the programmability [16, 17, 19, 43], and enabling over-subscription of the GPU memory (memory swapping) [21, 22, 28, 56].

However, on-demand paging requires the use of page faults (a type of exception) on the GPU which does not support the precise exceptions model [32, 46]. To overcome this limitation, GPUs offload all the page fault handling work to the CPU, while the faulted instruction on the GPU is stalled [58] (i.e., treated as a very long TLB miss). In this model, the GPU core is not even aware that the exception has occurred. Conversely, an exception on the CPU will switch the faulting thread into the exception handler routine which then saves the context of the thread, resolves the exception condition, and restores the execution of the thread. Throughout this paper we refer to the ability of preempting, and later restarting, the execution of a faulting thread as *preemptible* exceptions.

This ability to preempt on exception is used in many virtual memory features and optimizations in general-purpose systems [5, 20]. For instance, both lazy memory allocation and on-demand paging are implemented on top of this support. Whenever a page fault occurs, the OS exception handler checks if the cause is a page that has not been used yet (lazy allocation) or has been swapped out to disk (on-demand paging). In the former case, the OS allocates a physical page, updates the page table to reflect the new mapping, and restores the execution of the thread. In the latter case, the OS has to bring the page from disk, which is a very long latency operation. Hence, instead of waiting for the page to be brought to memory, the OS usually schedules a different thread to run, to maximize the system throughput.

Besides ensuring that exceptions are preemptible, most CPUs also guarantee the architecture state to be *precise* after an exception happens. The precise state is consistent with the state as if instructions were executed sequentially in program order, and execution stopped before the faulting instruction. This is commonly done by retiring instructions in program order (in-order commit) and buffering all the state changes caused by an instruction until it commits [18, 33, 46, 47].

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10...$15.00
https://doi.org/10.1145/3123939.3123950

Precise exceptions support thus allows a clean context switch and later restart of the faulting process.

Support for precise exceptions on GPUs, however, seems unfeasible. GPU cores are heavily multi-threaded, SIMD-like cores with very large register files (e.g., 256KB per GPU core in NVIDIA Pascal) and let the instructions commit out of the program order. Thus, the CPU techniques to implement in-order commit (e.g., the reorder buffer) would increase the area and power consumed by each GPU core significantly. Alternatively, it is possible to limit the execution model of the GPU core so that it commits instructions in-order, and therefore support precise exceptions, but this does not provide enough performance to meet the minimum requirements of modern GPUs. Instead, we argue for supporting imprecise but preemptible exceptions, as they can provide the functionality required by the system software (i.e., context switching) without sacrificing performance of the GPU core.

In this paper, we present three alternative implementations of preemptible exceptions for a modern GPU. We impose the minimal amount of execution constraints and track the minimal amount of additional state that provides us with an imprecise but well-defined restart point at which context switch can be performed. These three low-overhead design choices come with varying performance-complexity trade-offs. In our first design we ensure that no instruction from a warp is issued until all of its older instructions are guaranteed not to fault. This simple approach requires minimal changes to the GPU pipeline at the cost of decreased performance. In our second approach we relax this constraint with a mechanism that collects non-committed instructions for later replay. Such an approach results in slightly increased pipeline complexity, but also produces smaller execution overheads. Finally, our third solution introduces an operand log that allows aggressive score-boarding techniques to be applied, while still preserving replay ability. This approach further increases the area of the GPU core due to extra storage, but completely preserves the performance of our baseline GPU pipeline.

We further explore two use cases that aim to improve the system throughput thanks to the ability to context switch under a fault. In the first use case, we context switch out faulting threads during page faults and context switch in new threads from the same kernel, while the fault is being resolved. We aim to hide the latency of the page migration that caused the fault by finding other work that the GPU can execute in the meantime. In the second use case, we handle page faults to non-committed physical memory (i.e., lazy memory allocation) on the GPU itself, instead of off-loading it to the CPU. The GPU code runs its own physical memory allocator, which reserves the required memory and updates the GPU page table without interrupting the CPU.

The main contributions of this paper are:

(1) Three pipeline organizations, with different complexity-performance trade-offs, that allow the GPU to context switch on a page fault (Section 3).

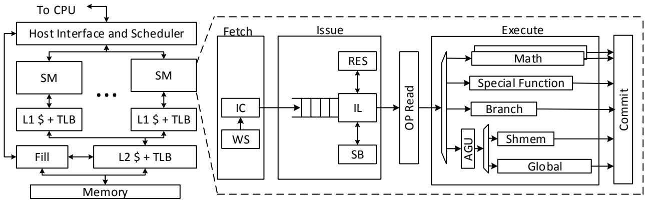


Figure 1: Baseline GPU architecture, with the SM pipeline zoomed in.

(2) A scheduling scheme that switches out faulted threads and runs other threads in their place to hide the latency of on-demand paging (Section 4.1).

(3) A fault handling scheme that handles some classes of page faults on the GPU itself, avoiding the expensive communication with the CPU (Section 4.2).

2 BACKGROUND AND MOTIVATION

In this section we first describe the GPU architecture used as our baseline throughout this paper. We use this baseline GPU to illustrate the main challenges we need to solve in order to support preemptible exceptions in a GPU. Finally, we motivate our work by discussing how features build on preemptible exceptions support can improve the GPU performance.

2.1 Baseline GPU Architecture

Figure 1 shows the baseline system architecture we assume in this paper. The host interface receives kernel launch commands from the CPU and partitions the launch into a number of independent tasks - *thread blocks* (TB). The thread block scheduler then starts issuing thread blocks to GPU cores - *Streaming Multiprocessors* (SMs) - for execution. Each SM has its own private cache memory and TLB, which are connected to a second-level shared cache memory and TLB respectively. Attached to the second-level TLB is a fill unit that performs GPU page table lookups when TLB misses occur.

We illustrate the detailed SM pipeline zoomed in Figure 1. The SM executes groups of thread blocks, which are further divided into warps (32 threads executing in the SIMT fashion [30]) that are interleaved on the SM. On each clock cycle, the warp scheduler (WS) picks one ready warp, for which an instruction line will be fetched from the instruction cache (IC) in the next cycle and sent to the issue stage. The resource allocation mechanism (RES) does resource bookkeeping (e.g., used register file banks), while the score-boarding (SB) mechanism enforces dependencies between instructions. Using this information, the issue logic (IL) schedules warp instructions in program order. Multiple instructions, from one or more warps, can be issued each cycle. The execution is non-speculative and the fetch for a warp is briefly disabled after fetching a control flow instruction, and re-enabling it at commit time.

Once issued, instructions go through the operand read stage, where the register file is accessed and the data is sent to the execution units. The execution stage consists of several math units, a branch unit, a special functions unit and different memory pipelines. The shared memory pipeline accesses the on-chip scratch-pad memory that holds CUDA *shared address space* objects, which are not subject to memory translation (each SM is used by only one user process at a time). The global memory pipeline performs the accesses to the off-chip memory that go through the cache hierarchy and memory translation. Upon completion, each instruction is sent to the commit stage, resulting in out-of-order commit.

Manipulating the score-boards is split between the operand read stage, in which source operands are released (after reading), and the commit stage, where the destination operand is released (after writing). Early release of the source operand score-boards helps minimize the *Write After Read* (WAR) hazards in absence of the register renaming hardware.

2.2 Current GPU Exceptions Support

Current generation GPUs support several types of exceptions which are treated differently. Exceptions like illegal instruction, division by zero, and floating point exceptions [44] normally result in the termination of the process. In these cases, graceful recovery from exceptions is not necessary, as they are only reported to the OS which requires modest exception support. GPUs also implement the *trap* instruction [2] that always causes an exception. Traps are handled by a routine running on the GPU, but since they do not require replaying any instructions, the kernel execution can be resumed easily.

Finally, the page faults caused by demand paging need special attention as they mandate that the faulting access is replayed, once the cause of the fault is removed. Current systems rely on offloading all the handling work to the CPU, while the faulting instruction is stalled in the middle of the pipeline [58], effectively being treated as a very long latency TLB miss. This approach to handling page faults guarantees correct execution but does not support preemption of the faulted instructions, as we describe in Section 2.5. Having non-preemptible faults in turn prevents the implementation of new virtual memory features and optimizations, including our two use cases.

2.3 GPU Demand Paging

Figure 2 illustrates the steps taken by the baseline system while performing on-demand page migration. When the GPU tries to access a page owned by the CPU, it misses in both private and shared TLB levels. The shared TLB forwards the miss to the fill unit, which, after walking through the GPU page table, determines that the page is not present in the GPU memory (1). The fill unit then sends a notification to the host interface, which in turn interrupts the CPU (2). The CPU interrupt handler (implemented by the GPU OS device driver) allocates the GPU physical memory (3), transfers the contents of the faulting page from the CPU memory to the GPU (4), and updates both CPU (5) and GPU (6) page tables to reflect the new location of the page. Had the GPU accessed a page that is not owned by the CPU nor GPU (i.e., first access to the page), the same steps would be performed, except for the data transfer (4) and the update of the CPU page table (5).

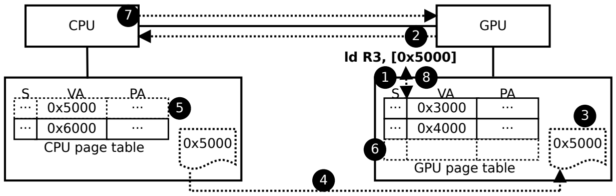


Figure 2: Demand paging in the baseline system.

Once the faulting condition has been resolved, the handler notifies the GPU (7) and the fill unit broadcasts this information to all SMs to resend the faulted requests [58]. Note that this only replays the memory request (from the microarchitectural state) and not the instruction itself. When the request is replayed, a valid translation exists, letting the faulting instruction continue execution (8).

2.4 Motivation

Traditionally, the page fault mechanism is used to implement many virtual memory features and optimizations, starting with the basic feature of demand-paging. Since some faults take a long time to resolve (if a page needs to be brought from disk), context switching to another process (or thread) became a goto solution to hide the latency of the fault and recover the throughput. Modern systems do not stop there, and come with a set of optimisations such as lazy allocation of physical memory, page prefetching, copy-on-write, etc. [7]. Operating systems also allow user level applications to handle faults [5, 14], facilitating further uses of the page fault mechanism in garbage collection [4], program checkpointing [29], and transactional memory [10].

The ability to squash and later replay faulting instructions is crucial to implement all of the above features. First, this way the forward progress of the handler routine (or any other thread, in case the OS chooses to perform a context switch) can be guaranteed because the pipeline gets flushed on a fault. Second, the preempted thread can be correctly restarted later, to resume the execution.

The implication of non-preemptible faults in our baseline system is that the handling routine has to run on the CPU, while the faulting instruction is stalled in the middle of the GPU pipeline. Offloading the fault to the CPU imposes a higher latency and lowers throughput of handling. Additionally, GPU threads with pending faults cannot be context switched (due to inability to correctly resume their execution later) which evidently hurts the systems throughput, but also causes other performance issues in multiprogrammed environments. As previous work has shown, a low context switch latency is the key to achieve good fairness and responsiveness in GPU multiprogramming [37, 50]. Therefore, the need for all the in-flight faults to be serviced before the context switch can happen increases the latency of context switching

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	R3 ← ld [R2]	F	I	O	E	E	E	E	E	E	E ❹								
B	R9 ← sub R9, 4		F	I	O	E	C ❶												
C	R8 ← ld [**R4**]			F	I	O ❷	E	E	E	E	E	E	E ❺						
D	**R4** ← add R7, 8				F	▨	I	O	E	C ❸									

Figure 3: Timeline showing the culprits of non-preemptible faults: sparse replay and RAW on replay. All instructions are from the same warp. Stages are Fetch, Issue, OpRead, Execute and Commit. Issue stall cycles are shaded dark gray.

significantly and possibly voids any effort to improve system performance through scheduling.

What makes this a performance critical problem is the massive amount of concurrent GPU threads (up to 32768 in the baseline). Even though faults are relatively infrequent events in the scope of a single thread, they become very frequent events in the scope of the whole GPU. The large amount of concurrent faults can overwhelm the CPU and the system interconnect that is used for both signaling and data transfers. During this time, faulted warps do not make any progress, and it is very likely that other warps (in the same, or different SM) are stalled on the same fault, resulting in a severely underutilized system. This lead us to propose two concrete use cases that can improve the system performance thanks to our support for preemptible faults.

The GPU programming model encourages programmers to oversubscribe the GPU with thread blocks in each kernel execution (i.e., request a number of thread blocks higher than the number of thread blocks the GPU can concurrently execute) as a way to keep the code performance-portable across the range of GPUs. Therefore, when a page migration is requested by the GPU, it is very likely that there are other pending thread blocks that could be executed. We detail the design of a scheme that context switches faulted thread blocks and tries to find some other, non-faulted blocks to run in their place in Section 4.1. This use case exercises the ability to preempt faulted threads and later restart them.

Even if the fault does not require a page migration from the CPU memory, the CPU is still performing the fault handling, as described in Section 2.3. This is indeed the case with faults that arise from the first use of a page (i.e., the CPU did not write to the page before). In Section 4.2 we detail the design of a system that, instead of offloading the fault handling to the CPU, handles faults on the SMs that have faulted, in the cases where it is feasible to do so. This use case exercises the ability to run the fault handler by the threads that faulted.

2.5 Problem Statement

To understand why exactly does the baseline pipeline prevent preemptible faults, let us consider the simplified pipeline operation in Figure 3. The oldest instruction in program order A goes through the fetch, issue, operand read stages and arrives to the global memory pipeline for execution. The global memory pipeline is deep, variable latency, and at some point (cycle 10 in the example) a page fault will be detected. The following instruction, B is independent so it is issued one cycle after A and commits normally (1). The third instruction C gets issued normally, and will also cause a page fault (5), just like the oldest instruction A. When instruction C reads the register R4 (the address of the load) in cycle 5, it releases the source operand score-board, signaling that there is no more WAR hazard between D and C (2). Finally, the youngest instruction D will issue after one cycle stall (due to previously active WAR hazard with instruction A) and commit (3), after writing the new value to register R4.

When instructions A and C trigger their faults in (4) and (5), they cannot be just squashed and later replayed from a saved architectural state (context). The first problem is that we have to replay several faulting instructions (the two loads A and C) but we must not replay instructions B and D, which have been already committed. However, no information is available in the pipeline to prevent the replay of instructions B and D. We refer to this problem as **sparse replay**.

The second problem is that the early source score-board release during the operand read stage allows the instruction D to write a new value to the register R4 at commit. Therefore, when we later replay instruction A, it will read the value in register R4 produced by the instruction D, leading to incorrect execution of the program. We refer to this problem as **RAW on replay**.

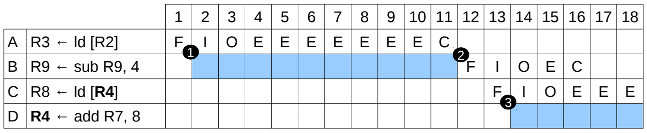
3 SUPPORT FOR GPU PAGE FAULTS

In this section we present three different approaches to support preemptible faults on our baseline GPU architecture. The first approach (*warp disable*) treats memory instructions as code barriers, making it easy to replay the faulting one. The second approach (*replay queue*) introduces a replay queue to track in-flight memory instructions (part of the context). Finally, the third approach (*operand log*) also logs the source operands of memory instructions, enabling the aggressive score-boarding techniques. Each of these approaches presents a different trade-off between the amount of ILP the architecture exploits and the additional hardware needed.

3.1 Approach 1: Warp Disable

This scheme addresses both sparse replay and RAW on replay problems defined in Section 2.5 by treating global memory instructions (i.e., the only instructions that can potentially page fault) as an instruction barrier. We enforce this by disabling the warp fetch once a global memory instruction is fetched and re-enabling it once the instruction commits. Note that execution of other warps is not affected by the events in a single warp. By the time the instruction is ready to commit it had finished all the work, including the TLB accesses for all the active threads. Thus, at commit time we can guarantee that the memory instruction will not fault and necessitate a replay. If the fault does occur, the limitation of the model has provided us with two benefits:

- It is guaranteed that only one of the warp in-flight instruction can fault.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	R3 ← ld [R2]	F ❶	I	O	E	E	E	E	E	E	E	C ❷							
B	R9 ← sub R9, 4		🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	F	I	O	E	C		
C	R8 ← ld [**R4**]													F ❸	I	O	E	E	E
D	**R4** ← add R7, 8														🟦	🟦	🟦	🟦	🟦

Figure 4: Pipeline timing diagram with the warp disable approach. Global memory instruction disables fetching new instructions for the warp (shaded in light blue).

- In the case of the fault, it is always the last fetched and issued instruction for the warp that faulted.

Hence, we only need to squash and later replay that one faulted instruction.

The pipeline timing diagram in Figure 4 illustrates how *warp disable* works. After fetching a potentially faulting instruction A, the SM stops fetching any new instructions from this warp (1). If instruction A finishes successfully, the fetch is enabled again (2), letting the younger instructions B and C execute. Since C is also a potentially faulting instruction, it will disable the warp fetch again (3).

If any of the potentially faulting instructions (A and C) faults, the only other instructions that could be in the pipeline are older instructions that never cause a page fault. To recover from the fault, we squash the faulting instruction and drain all other in-flight instructions of the warp, before invoking the exception handler. To restart the execution, the exception handler restores the program counter to the instruction that caused the exception, for it to be replayed.

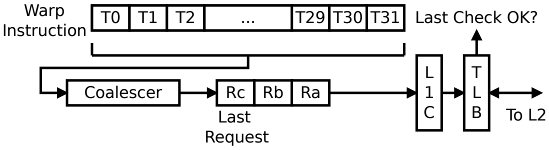
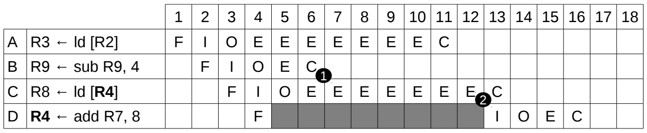


Figure 5: Last TLB check for a warp memory instruction: the earliest point in the pipeline where memory instruction is guaranteed not to cause a page fault.

We can further optimize the performance of this scheme by realising that in all cases, we could enable the warp before the commit stage. Because a warp consists of 32 threads, one memory instruction of a warp can be accessing multiple pages at the same time. As shown in Figure 5, the instruction first goes through the coalescing unit that generates one memory request for each unique cache line accessed by the warp (part of the baseline SM). The earliest cycle where we can re-enable the warp so that it continues fetching and issuing instructions is right after the TLB check for the last generated request has completed successfully. The result of moving fetch-enable to the earliest cycle possible is letting other instructions enter the pipeline as soon as possible.

This scheme is also applicable to other types of exceptions, such as *divide-by-zero*, by treating the instructions that may trigger the exception as code barriers. Analogously, we enable the fetch for the warp as soon as we can ensure that the instruction is not faulting.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	R3 ← ld [R2]	F	I	O	E	E	E	E	E	E	E	C							
B	R9 ← sub R9, 4		F	I	O	E	C ❶												
C	R8 ← ld [**R4**]			F	I	O	E	E	E	E	E	E	E ❷	C					
D	**R4** ← add R7, 8				F	▨	▨	▨	▨	▨	▨	▨	▨	I	O	E	C		

Figure 6: Pipeline timing diagram with the replay queue approach.

The negative side of the warp disable scheme is that it hinders the ILP achieved by the baseline SM by temporarily disabling the warp on a memory instruction. Since the SM is a throughput oriented processor that heavily relies on multithreading to achieve high performance, we can expect smaller performance impact than a similar technique would have on a CPU. Furthermore, since other instruction types cannot page fault, their execution is unchanged from the original SM. The positive side is that we have enabled preemptible faults without increasing the hardware complexity.

3.2 Approach 2: Replay Queue

Our goal with this scheme is to remove the instruction barrier semantics imposed by the warp disable scheme, so that SMs can exploit larger amounts of ILP. To deal with the RAW on replay problem, we implement a more restrictive scoreboarding scheme, compared to the baseline system. Here we release source operands of global memory instructions only after the last TLB check has completed successfully, while other instructions are handled like in the baseline pipeline (i.e., releasing source operands in the operand-read stage).

After handling the RAW on replay problem, we are still left with the problem of the sparse replay, as shown in the pipeline timing diagram for our example program in Figure 6. Instructions A, B, C are fetched and issued back to back, while the youngest instruction D has to be stalled over a WAR hazard on register R4. B executes normally and commits a few cycles later (1). If instruction C does not causes a fault, the source score-board for register R4 will be released (2) and instruction D continue with normal execution. However, if one or both of the load instruction do fault, we must ensure to replay them when the fault is resolved, but the committed instruction B must not be replayed.

To deal with the sparse replay problem we add a replay queue next to the issue queue. Global memory instructions (the only ones that can cause a page fault) are inserted in the replay queue when issued, and removed once they commit. Because the replay queue holds only potentially faulting instructions, its size is bounded by the number of in-flight global memory instructions. If a fault is raised, we first drain all the non-faulted in-flight (already issued) instructions, squash all the faulted ones, and revert the program counter to the oldest non-issued instruction. The instructions in the replay queue now become part of the context and need to be saved during a context switch. When the execution of the warp is restored, the saved instructions are replayed

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	R3 ← ld [R2]	F	I	O	E	E	E	E	E	E	E	C							
B	R9 ← sub R9, 4		F	I	O	E	C												
C	R8 ← ld [**R4**]			F	I	O	E	E	E	E	E	E	E	C					
D	**R4** ← add R7, 8				F	▨ ❶	I	O	E	C									

Figure 7: Pipeline timing diagram with the operand logging approach.

first, before we continue with normal execution. Because the replay instructions are captured in the program order (relative to each other), the baseline issue logic and scoreboard mechanisms can guarantee correct execution. Similar to our related work, context save and restore can be performed in software [37, 50] or hardware [57].

Other types of exceptions could be also supported with these scheme, through simple extension. The source operands of instructions that can possibly cause an exception must be released only after making sure that they will not raise an exception. Additionally, these instructions must be inserted in the replay queue when issued and removed at commit.

The negative side of the replay queue scheme is that it introduces additional complexity to the baseline system. This is reflected in both additional hardware (the replay queue) and a more complex fault handler routine, because the contents of the queue are also part of the context. The positive side is that the replay queue is an unobtrusive addition that improves the ILP over the warp disable scheme by eliminating the barrier instruction semantics. Furthermore, the replay queue does not hold any data (i.e., registers) produced or consumed by the instructions, an important property bearing in mind that a warp instruction is a 32-wide SIMD instruction.

3.3 Approach 3: Operand Log

Our goal with this scheme is to improve even further the performance of the replay queue scheme by removing the conservative release of global memory instructions' source operands after the last TLB check has completed successfully. Instead, we want to perform all the score-boarding operations at the operand-read stage for all instructions, like in the baseline GPU. Let us consider the pipeline timing diagram for our example program in Figure 7. Instructions A, B, C are fetched and issued back to back, while instruction D has to be stalled over a WAR hazard on register R4. The WAR hazard on register R4 is removed when instruction C reads its value (1), and instruction D gets issued in the next cycle. By the time any of the potential faults are raised, instructions B and D have already committed. If instruction C faults and needs to be replayed, it would read the wrong (updated) value from R4.

To handle the RAW on replay problem, we augment the SM with an operand log that holds the source data of in-flight global memory instructions (the only ones that can cause a page fault). Note that the operand log only eliminates the RAW on replay problem, so we still need the replay queue to address the sparse replay problem.

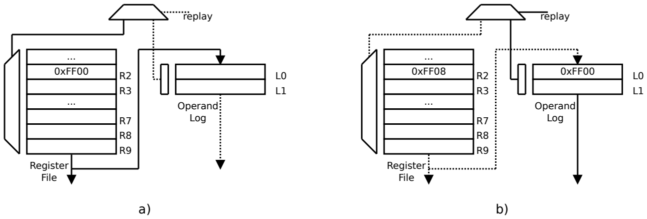


Figure 8: Design of the operand log with active path during a) first issue and b) replay of a faulted instruction.

The allocation of the log entries is performed during issue of a memory instruction. During the operand read stage, data read from the register file is written to the log, making it safe to release the source operands, since now there is a copy that can be used in case of replay. To optimize the use of log space, load instructions take up only one log entry (source address), while store instructions take two (source data and destination address). Entries are released once we know the instruction is not faulting (after the last TLB check for the instruction has completed). On replay, the instruction reads the source operands from the log instead of the register file. In our implementation we assume log implementation with a single-ported SRAM memory, because the SM can issue only one memory instruction per cycle.

Just like the contents of the replay queue, the log is now also part of the context and needs to be saved and restored during context switch. Since we need to provide the context switching at a thread block granularity, the log is partitioned (at the kernel launch time) so that each running block gets its own partition. Thus, kernels with lower number of active thread blocks (SM occupancy) will have higher number of log entires per thread block, and vice versa.

The negative side of the operand log scheme is that it introduces further hardware overheads, and because of the increased context size, it causes a higher context saving and restoring latency. The positive side is that now both culprits of preemptible faults (sparse replay and RAW on replay) are eliminated, achieving the performance of the baseline SM while enabling preemptible faults (with a sufficiently large log).

4 USE CASES

In this section we present two use cases that improve performance of GPU systems with demand paging, but require GPU support for preemptible page faults. In the first usecase we context switch the SM when a page fault triggers an on-demand page migration from the CPU, which is a long latency operation. In the second use case, we rely on page faults to perform on-demand allocation of physical memory on the GPU.

4.1 Block Switching on Fault

Although the SM can issue other instructions that do not depend on the faulting memory access, oftentimes the pool of available independent instructions gets exhausted much earlier than the faulting memory access completes. Hence, the SM sits idle waiting for the fault to be resolved, underutilizing the available hardware resources. The preemptible exception support we present in this paper opens the door to context switching the SM when a page fault happens, so that some other threads can use the SM resources while the fault is being resolved. The key observation for this proposal is that the programming model of the GPU maximizes the likelihood of finding such threads in the pool of pending thread blocks.

The GPU kernels are normally launched to execute a very large number of thread blocks. Each thread block is a group of threads that are executed concurrently, on the same SM, and can communicate and synchronize with other threads of the same block. The number of thread blocks that can execute concurrently depends on the available SM resources (e.g., size of the register file and shared memory) and number of SMs, which are specific for each GPU architecture and model.

The GPU programming model encourages programmers to launch a much larger number of thread blocks than the GPU can concurrently execute. Because newer GPUs are capable of concurrently executing larger number of thread blocks [35], oversubscribing the GPU is intended to preserve the application scalability. Upon the kernel launch, an initial batch of thread blocks that fills the whole GPU is issued. Pending blocks are then issued only when one of the running blocks finishes execution. It is therefore likely that there are still pending blocks when a page fault occurs.

Due to the possibility of synchronization and communication of threads inside the block, granularity of context switching in this scheme has to be on a thread block level. The state of a thread block includes the contents of its shared memory partition and register file (for all of its threads), and control information such is barrier unit state, etc. If the SM implements the replay-queue or operand-log schemes, entries associated with a thread block are also part of its state. When blocks are switched out, their state is kept in a preallocated GPU memory area, similar to recent multiprogramming proposals [37, 50] and CUDA dynamic parallelism (i.e., kernel launch from the GPU itself) feature [35].

To implement the block switching scheme, we introduce the notion of *off-chip* blocks and augment the SM with a *local scheduler*. The local scheduler tracks the state of *active* blocks (running, with all of the context in the SM) and *off-chip* blocks (preempted, with all of the context in memory). When an SM faults, it gets a notification from the fill unit (MMU), including the position of the fault in the global pending faults queue (maintained by the baseline fill unit). To avoid wasteful context switching, the local scheduler will decide to switch out a block only if this position is above a set threshold.

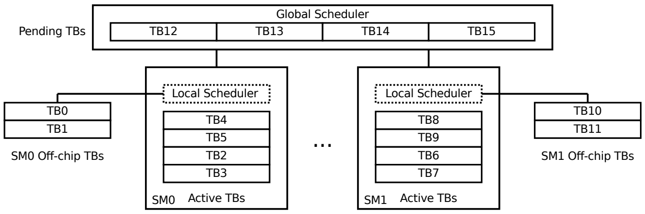


Figure 9: Thread block switching scheme: local scheduler makes context switching decisions and keeps track of SM's active blocks (context on the chip) and off-chip blocks (context in memory).

After switching out a block, the local scheduler has to decide what to run on the newly freed resources. If there are off-chip blocks with no pending faults (all have been resolved) it will restore the context of one of them (promoted to active block) and let it continue with execution. If there are none, the SM requests a new pending thread block from the global scheduler, in a similar way the SM currently does whenever a thread block finishes its execution. To prevent explosion of the off-chip memory space used for the thread block contexts, the local scheduler is allowed to bring only a limited number of extra blocks to the SM (4 additional blocks in our configuration). Once this limit is reached, the SM cycles through the active and off-chip blocks only.

4.2 Local Handling of Faults

When a page fault in the GPU occurs on a page that has not been previously accessed by the CPU (first use of the page), no page migration from the CPU memory is required. However, in our baseline system the CPU is still in charge of managing both CPU and GPU page tables and physical address spaces. Hence, the CPU still needs to be interrupted to perform the allocation of the physical memory on the GPU and update the GPU page table before the faulted instruction can continue. There are several cases that lead to such faults on the GPU, including the pages that hold the output data of the kernel or the pages that are backing a heap memory allocation (i.e., malloc) performed by the kernel itself.

The preemptible exception support we introduce in this paper allows us to run a page fault handler in the GPU that will perform physical memory allocation and page table management. When a warp faults, it is switched to system mode [2] and executes the handler which checks the faulting address and determines whether it corresponds to memory owned by the CPU, has no physical memory assigned, or is an invalid memory access. If the page is owned by the CPU, the handler sends the data migration request to the CPU. If the address is an invalid memory access, it sends a request to the CPU to abort the kernel execution. Finally, if the address has not been assigned any physical memory yet, it marks the page as owned by the GPU (to prevent the CPU to allocate memory for this page), allocates a new physical memory page on the GPU, updates the GPU page table, and restarts the execution of the faulted warp.

This concurrent memory management requires a consistency protocol to guarantee correctness. The prototype system that we based our evaluation on employs a relaxed consistency protocol to minimize the overheads. Fine-grained synchronization and address space (both virtual and physical) partitioning techniques are used to minimise the contention when accessing shared structures. Thread safety on the GPU level is achieved using lock free data structures and atomic memory operations, while synchronization on the system level is performed using Szymanski's algorithm [49].

5 EXPERIMENTAL EVALUATION

In this section we first compare the performance of the three pipeline designs proposed in Section 3 to the baseline pipeline. We demonstrate that our most aggressive scheme, operand log, can achieve the baseline performance with a modest log size, while the other two schemes offer significant performance at lower complexity, in most cases. We also show how the use cases proposed in Section 4 improve the performance thanks to the ability to context switch.

5.1 Methodology

The results in this paper are obtained using an in-house developed cycle-level timing simulator that models the GPU architecture presented in Section 2. It consumes dynamic instruction and memory traces generated by an execution-driven functional simulator. The timing simulator models detailed SMs, cache hierarchy and MMU (TLBs and fill unit) attached to a simple DRAM model. We use an ISA designed to mimic modern GPU ISAs with all the distinguishing features such as a large unified RF, explicit management of the divergence stack, fused-multiply-add instruction, approximate complex math instructions, etc. Benchmarks are compiled from their CUDA sources using NVIDIA NVCC 5.0 to generate the LLVM [27] intermediate representation (IR) assembly for all the kernels. The LLVM IR is then compiled to the target ISA by our compiler backend built on LLVM version 3.3. The parameters of the baseline GPU, given in Table 1, are based on the NVIDIA Kepler K20 GPU with 16 SMs. We use memory hierarchy parameters that are aligned with the experimentally measured data reported in literature [6].

We use all the benchmarks from the Parboil benchmark suite [48] in our evaluation. Additionally, in Section 5.4, we use the Halloc benchmarks [1] and one CUDA SDK sample to evaluate our proposals on local page fault handling. We simulate one kernel from each benchmark to its completion. If benchmarks have multiple kernels or multiple launches of the same kernel, we simulate the main kernel execution.In the experiments that perform on-demand page migration, all data is initially residing in the CPU memory. Even though our proposals are compatible with oversubscription of the GPU memory (i.e., memory swapping), we do not evaluate it since none of the simulated benchmarks have the dataset large enough to cause it.

SM:	
Frequency	1GHz
Max TBs	16
Max Warps	64
Register File	256KB
Shared memory	32KB
Issue ways	2 instrcutions total from 1 or 2 warps
Backend units	2 math, 1 special func, 1 ld/st, 1 branch
L1 cache	32KB / 4-way LRU / 128B line 32 MSHRs / 40 clk latency / virtual
L1 TLB	32 entires / 8-way LRU
System:	
Number of SMs	16
L2 cahce	2MB / 8-way LRU / 128B line 70 clk latency / 512 MSHRs
L2 TLB	1024 entries / 8-way LRU 128 MSHRs / 70 clk latency
Numer of PT walkers	64
Walking latency	500 clk
DRAM bandwidth	256 GB/s
DRAM latency	200 clk

Table 1: Simulation parameters used in the evaluation.

We assume 4KB [38, 39, 58] GPU pages. Related work [58] and our own experiments indicate that some form of prefetching is necessary to make the on-demand paging competitive in performance. Thus, when evaluating the use cases (Section 5.3 and Section 5.4) we do handling with a 64KB granularity. This helps to amortize the high cost per fault caused by communication, system software and inefficient small data transfers. When evaluating use cases, we start with a baseline that already supports preemptible faults (with replay queue), to measure the isolated benefit of each technique.

5.2 The Performance Cost of Preemptible Faults

The different pipeline organizations with support for preemptible faults that we presented in Section 3 are by design expected to have different performance. We compare their performance to a baseline SM that, owing to the disregard of preemptible exceptions, represent the maximum performance our proposals can achieve. In Figure 10 we show the performance of two *warp disable* scheme variants described in Section 3.1 (warp disable until commit - *wd-commit* and warp disable until last TLB check - *wd-lastcheck*), alongside the performance of the *replay queue* scheme described in Section 3.2, all normalized to the baseline SM. We are here foremost interested in the performance of kernel execution without any faults (e.g., expert written program that uses explicit data management). Such execution will show us exactly how much performance loss is caused by our pipeline changes.

Comparing the geometric mean performance achieved across all benchmarks, we can see that WD-commit achieves only 84% of the baseline performance while WD-lastcheck achieves 90% of the baseline performance. The difference between these two schemes is related to how early in the pipeline we re-enable warp fetch. This results show that with a simple modification to the warp disable scheme (WD-lastcheck), we are able to recover significant amount of performance. The replay queue scheme is able to close this gap further, achieving

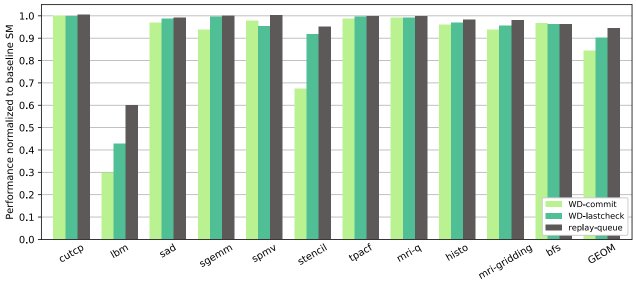


Figure 10: Performance of *warp disable* and *replay queue* pipeline organization that support preemptible faults, normalized to the baseline SM with stall-on-fault approach (higher is better).

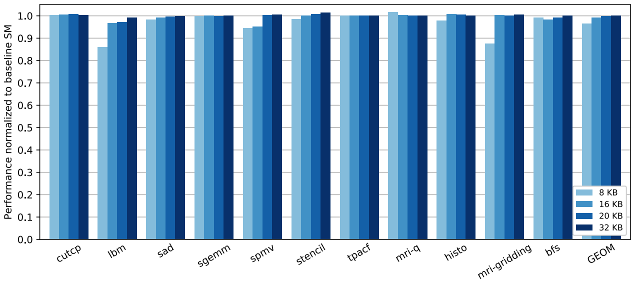


Figure 11: Performance of the operand log scheme with various log sizes, normalized to the baseline SM with stall-on-fault approach (higher is better).

94% of the baseline performance. There are a few cases where even the replay queue scheme is not sufficient, especially *lbm* that achieves only 60% of the baseline performance.

The *operand log* scheme presented in Section 3.3 is designed so that it can achieve the performance of the baseline SM if a sufficiently large log is used. In order to find out what is a sufficiently large log size, we show the normalized performance of the operand log scheme with various log sizes in Figure 11. We start exploring log sizes from 8KB because it is the smallest log that guarantees that all thread blocks of a kernel with maximum occupancy (i.e., 16 in our baseline system) can execute concurrently. Indeed, the biggest amount of source data that an instruction needs to log is 512B, assuming 8B address and 8B data (times the 32 threads). Thus, such log guarantees that each thread block can have at least one memory instruction in case that the SM occupancy is 16 thread blocks. Comparing the geometric mean performance achieved across all benchmarks, we can see that even the 8KB log is capable of achieving 96.6% of the baseline performance, while the 16KB log is capable of achieving 99.2%. The operand log scheme is the most effective with the *lbm* benchmark, where the 16KB log improves the performance from 60% to 97% of the baseline, compared to the replay queue scheme.

The effectiveness of the proposed schemes varies depending on the benchmark. Those with a high level of TLP do not have any performance improvement with the different schemes, as they already benefit from the GPU's latency hiding abilities, even with the simplest warp-disable approach. More advanced schemes, like the operand log, are most compelling with difficult codes that exhibit insufficient parallelism to saturate the modern GPU. The most evident case is the *lbm* kernel that, due to the large amount of registers used by each thread, runs at a low SM occupancy of only 8 warps (one eighth of the total supported by the SM). As a result, the increased ILP of the replay queue and operand log schemes leads to high performance improvements over the warp-disable scheme, almost reaching the baseline performance with a 32 KB operand log.

We also analyzed the area and power overhead of our proposals using CACTI [34] version 6.5. To study the area overhead, we use the methodology and baseline numbers reported in [40], while for power overhead we use the high-level power model and baseline numbers reported in [15]. Since both [40] and [15] discussions are in the context of $40nm$ technology, we use the same node size to model the operand log in CACTI. We also apply a factor of 1.5 to account for extra overheads like the control logic. For area comparison, we use a conservative estimate of the GPU area ($561mm^2$ for a chip with 16 SMs) and the SM area ($16mm^2$) [40]. To put the power overheads in context, we assume a SM power consumption of 5.7W and a total GPU power (chip only) of 130W [15]. We have used CACTI to obtain operand log leakage power and energy of one access, and computed the total power assuming the worst case scenario of one log write per cycle. Table 2 shows the relative area and power overheads of the operand log scheme. For all log sizes except the largest studied (32 KB), the total GPU overheads are below 1% area and 2% power.

Log Size	SM Area	GPU Area	SM Power	GPU Power
8 KB	1.04%	0.47%	1.82%	1.28%
16 KB	1.47%	0.67%	2.34%	1.64%
20 KB	1.67%	0.76%	2.61%	1.83%
32 KB	2.36%	1.08%	3.38%	2.37%

Table 2: Operand logging overheads.

5.3 Use Case 1: Block Switching on Fault

We show the performance of thread block switching scheme in Figure 12, for NVLink and PCI express 3.0 interconnects. We have measured several principal components that add up to the round trip latency of a page fault (page pinning, physical page allocation and the data transfer itself) and combined them with the interconnect latencies to compute the cost of a page fault. We estimate the separate costs of faults for the case when there is a data transfer and for the case when only the allocation is necessary (pages not dirty in the CPU page table). These estimates are $12\mu s$ /$10\mu s$ for NVLink and $25\mu s$ /$12\mu s$ for PCIe, respectively. We have setup the local scheduler to allow a maximum of 4 extra thread blocks per SM. For each interconnect, the execution time is normalized to the on-demand paging implementation that does not switch blocks on fault.

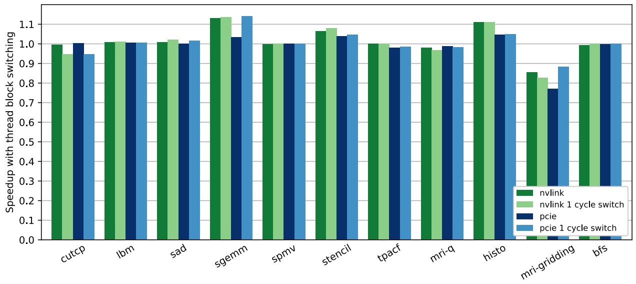


Figure 12: Performance improvement with thread block switching on a fault over a system with no switching. Showing NVLink and PCIe configurations with normal context switching and ideal 1 cycle context switching.

Starting from the NVLink, we can observe that several benchmarks show a notable performance improvement. These are *sgemm* with 13%, *stencil* with 7% and *histo* with 11%. With the PCIe interconnect, the same benchmarks exhibit performance improvement, albeit a lower one (*histo* is the highest with 5%). We also show the performance of this scheme with ideal context switching (1 cycle save and 1 cycle restore). Notice that in most cases there are small or no improvements. Performance comparison with ideal context switching demonstrates how our local scheduler is doing a good job on avoiding unnecessary context switching. It also shows that we captured most of the performance improvement that can be achieved through increasing the set of active thread blocks.

We have studied this performance further, and found out that out of 11 benchmarks, 5 have either a very low or very high interconnect utilization. Thus, any scheme that tries to overlap computation with transfers is not going to improve performance on these benchmarks. From the rest of the benchmarks, 3 have either unfavorable access patterns such is faulting at the end of the block, or suffer from a severe tail effect. The performance degradation of *mri-gridding* counters the improvement of other benchmarks, resulting in unchanged average performance.

It is important to note that no benchmark has a notable performance degradation except *mri-gridding* which achieves 85% of the original performance due to the massive load imbalance that the kernel exhibits. In this benchmark there is a two orders of magnitude difference in thread block execution time, owing to the different amount of work performed by thread blocks. We have traced the execution and noticed that the original thread block distribution in our configuration (16 SMs) happens to almost evenly spread the longest blocks across the SMs. Once context switching starts changing this order, most SMs finish faster due improved latency hiding, but a minority of SMs get penalized with extra long blocks. Since we measure the execution time of the kernel as the cycle when the last thread block finishes, this ultimately leads to longer execution. This is further evident from *mri-gridding* performance with ideal context switching being lower than with normal context switching in NVLink configuration.

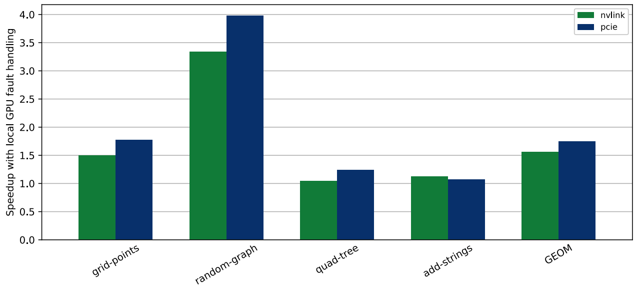


Figure 13: Performance improvement when handling faults to pages that are backing up dynamically allocated memory on the GPU over baseline handling by the CPU.

5.4 Use Case 2: Local Handling of Faults

In Section 4.2 we have described a scheme that allows handling page faults on the GPU itself, if the data transfer from the CPU is not required. The prime example of this are pages that are backing up memory allocations performed by the kernel itself (e.g., through the CUDA device version of malloc). Since Parboil kernels do not use device side malloc, we evaluate the performance using the benchmarks that ship with the Halloc CUDA dynamic memory allocator [1]. Additionally, we have ported one of the CUDA SDK sample applications (*quad-tree*) to use dynamic memory allocation (each node allocates its children dynamically instead of allocating all the possible nodes beforehand, given a maximum depth) and removed the dynamic kernel invocations (simulator limitations). There is no page migrations in this experiment (i.e., explicit transfers), and all the page faults are caused by accesses to unmapped pages (first use). We measured the performance and scalability of a prototype fault handler on a real GPU and assume the latency of the GPU handler to be 20μs, an order of magnitude more than the estimated latency of the CPU handler (2μs) used in the rest of this section.

Figure 13 shows the performance improvement with a geometric mean speedup of 56% and 75% for NVLink and PCIe, respectively. The reason for such performance improvement, considering the higher latency of handling, lies in the number of concurrent page faults. The GPU is running many threads concurrently, and even though the frequency of fault in each thread might be low, the large working set of a GPU produces enough faults to overwhelm the system interconnect and the CPU that have to handle them one by one. In contrast, handling them on the GPU results in a clear throughput win, despite the longer latency of each fault.

In Figure 14 we show the performance of handling the faults to output pages caused by Parboil kernels. These pages hold the output data of the kernel, and are not accessed by the CPU until the execution of the kernel finishes. Benchmarks like *lbm* and *histo* show significant performance increase in both configurations. Contrary to the results in Section 5.3,

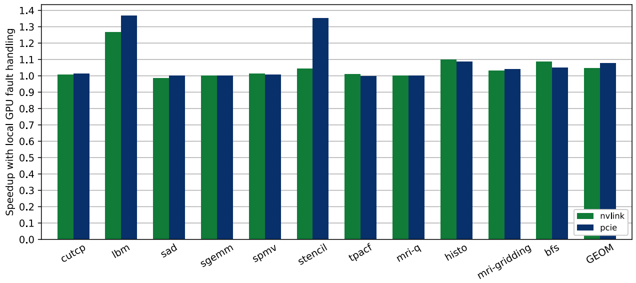


Figure 14: Performance improvement when handling faults to output pages on GPU over baseline handling by the CPU.

this time the PCIe configuration shows an overall higher performance improvement than the NVLink configuration. Geometric mean across all benchmarks for NVLink is 5%, and for PCIe is 8%. A higher performance improvement is seen with PCIe because the higher fault cost compared to the NVLink leads to higher contention of the system interconnect.

5.5 Scalability Discussion

As GPUs ship in a wide range of configurations (i.e., number of SMs), it is also interesting to observe the impact of GPU size on the performance of our proposals. When the amount of concurrent work performed by the SM stays the same (i.e., running at the maximum occupancy), the three SM pipeline designs exhibit the same performance trends as reported in the Section 5.2. However, if the workload size does not scale with the size of the GPU (decreased SM occupancy), the performance gap between the three schemes widens further. The number of SMs plays a more significant role in the performance of the two use cases. Increasing it leads to a higher number of concurrent page faults and higher contention of the system interconnect. Performance with block switching (use case 1) depends on the compute/transfer ratio of the benchmark. Increasing the number of SMs will lead to lower performance on transfer-bound benchmarks and increased performance on compute-bound benchmarks with favorable access pattern. Finally, when handling faults locally (use case 2), the performance improves with the number of SMs, because local fault handling decreases the contention of the CPU and the interconnect. We perform the evaluation with a common 16 SMs configuration, because the Parboil benchmarks were written for GPUs of that size and many of them do not scale to larger GPUs.

6 RELATED WORK

Early pipelined processors did not fully support precise exceptions. For example, IBM System/360 model 91 [3] had a floating point unit (scheduled using Tomasulo's algorithm [52]) that was causing imprecise exceptions. Other high performance computers like the CDC 6600 [51] and Cray Research Cray-1 [42] vector machine implemented out-of-order commit, thus they did not support precise interrupts, and virtual memory for that matter.

Modern general-purpose CPUs support precise exceptions and usually handle different types of exceptions (external interrupts, faults and traps) the same way. Smith et al. [46] discuss three mechanisms to recover from an exception in a precise manner: reorder buffer, history file, and future file. Hwu et al. [18] proposed the checkpoint-repair mechanism while Sohi [47] and Moudgill et al. [33] discuss use of the unified register file that holds architectural and speculative state. All of these proposals are focused on supporting exception recovery to *precise* architectural state by significantly increasing the storage to include architectural and speculative state. Thus, they can also be used as a misprediction recovery mechanism. This large increase of state is not practical in GPUs due to the already very large register file. Furthermore, the expected benefit of speculative execution does not justify the added complexity.

Implementing exceptions in vector processors was proven to be a challenging task, also. For that reason, many vector machines omitted the support for virtual memory [24, 42, 54]. IBM System/370 allowed only one vector instruction to be in-flight [9], which simplifies the support for exceptions, but limits the performance. Several vector processors / extensions have been proposed in academia with support for precise [13] (using the reorder buffer approach) or restartable exceptions [25] (using the history file approach). Tarantula was a vector extension for the canceled Alpha EV8 processor [12] that supported precise exceptions by piggybacking on the host EV8 renaming capabilities.

As a way of handling exceptions in exposed pipeline processors, Rudd [41] proposed redirecting the output of the pipelines into a replay buffer instead of feeding them to the write back stage. Sentinel scheduling [31] is a compiler technique for detecting exceptions of speculatively scheduled instructions in VLIW processors. It is focused on correct signaling only, i.e., not restarting the process after the exception is handled. To tackle the restartability issue, Bringmann et al. [8] proposed a write-back suppression scheme as a method of recovery from speculatively scheduled instructions in VLIW processors. Both replay queue and write back suppression schemes perform result buffering akin to that of the reorder buffer from [46]. In contrast to these, our operand log scheme from Section 3.3 is our only scheme that does data buffering. We only buffer the data for global memory instructions, and we only do it until the instruction passes TLB checks. Thus we are minimizing the state buffering overhead, while maximizing the performance by still allowing out of order commit.

The *invisible exchange* package solution used in the CDC CYBER 200 machines did a snapshot of all the microarchitectural state, which was saved and restored as part of the context [46]. In the case of GPUs, this would entail directly manipulating the state of the warp scheduler, score boards, SIMT divergence hardware, load store unit (that holds dozens of coalesced memory requests), etc. Doing a snapshot of the issue queue and saving it as a part of the context was proposed by Torng et al. [53]. It reminds of our replay queue scheme, except that we do not keep the issued instructions in the issue queue. Instead, our replay queue captures only the ones that need to be replayed, allowing for a smaller and simpler issue queue.

There have been several proposals for exception handling in multi threaded processors [23, 59], that stall the faulting instruction, while handling the exception in the execution thread. The goal of this approach is to start executing the handler code as soon as possible by avoiding the instruction flushing, state repairing and context switching latencies. These approaches still need the hardware support for precise exceptions to allow process restart, in case that the handler decides that context switch is needed, after all.

Kruijf et al. [11] proposed a processor design that executes idempotent regions of the program (regions that can be executed multiple times with the same result) constructed by the compiler. iGPU [32] is an application of idempotent processing to GPUs to allow recovery from exceptions. This approach introduces runtime overheads due to the additional instructions generated by the compiler (register spills that perform state preservation). Compared to this, the mechanisms for fault recovery and handling proposed in this paper do not require compiler support. Furthermore, with sufficient amount of resources, our operand log scheme is able to achieve the performance of the baseline pipeline. In Section 5.2 we show how even a relatively small log can achieve this performance.

Some of the previous work on GPU multiprogramming utilizes a context switching mechanism [37, 50, 57]. Because the scheduler interrupt can be delayed, a common technique they use is pipeline drain (including all the in-flight memory instructions), before the architectural state is saved off-chip. None of these proposals assume presence of page faults that can render the kernel unpreemptible for a long time. Proposals for preemptible exceptions presented in Section 3 enable their schedulers to also perform fast context switches of faulted kernels. Shahar et al. [45] argue for page fault handling on GPUs, in order to enable their paging techniques and demonstrate performance improvement with applications that oversubscribe the GPU memory. Because of the limitations of current hardware, they propose a software layer that performs address translation. Our proposals eliminate the need of an intermediate software layer and utilize the hardware TLBs provided by the GPU, while still providing them with the ability to implement their paging technique in the fault handler routine. Finally, Vesely et al. [55] studied the performance of virtual memory on the AMD APU, a fused CPU-GPU SoC, and proposed directions for further improvements. Among other things, they observed that faults caused by the GPU have much higher latency and much lower scalability of handling than faults caused by the CPU. Our baseline system exaggerate this even further because of the latency and bandwidth limitations of the PCIe/NVLink system interconnect used to connect CPUs with discrete GPUs. Support for allocation of physical memory by the GPU fault handler outlined in this paper relieves the stress put on the link-CPU chain and improves the performance through scalable fault handling.

7 CONCLUSIONS

In this paper we have presented three different approaches to support preemptible exceptions on modern GPU architectures. The proposed approaches pose different trade-offs in terms of hardware requirements and performance overhead introduced over the baseline architecture. We show that these approaches can achieve 90% of the baseline performance with no added complexity and 99% of the performance with relatively small increase in the area.

We have also explored two potential use cases for exceptions on modern GPUs: context switching during page migrations and local fault handling for lazy physical memory allocations. Although context switching produces performance improvements only in select circumstances, it boosts the performance of two of the most common GPU codes: *sgemm* and *stencil*. This performance improvement is likely to benefit a large number of applications, ranging from physical simulations to linear algebra solvers.

The performance of lazy physical memory allocation for output data pages is also encouraging. However, being able to apply this technique to dynamically allocated GPU memory (i.e., malloc inside a kernel) greatly improves its usefulness. Without the ability to efficiently allocate physical memory on demand, current implementations of malloc statically allocate large portions of GPU physical memory at the application load time. This effectively reduces the available memory on the GPU for applications and, thus, most programmers avoid using malloc. By allowing malloc to only consume those physical memory actually required, we expect this functionality to be more widely used.

Besides the use cases we have discussed, the exception support in the GPU we have presented in this paper opens the door to further facilities provided by the operating system in the CPU to become available to GPU codes. This would increase the number of applications suitable to be accelerated by GPUs and, overall, improve the programmability of such systems.

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers, Lluis Vilanova and Javier Cabezas for their help in improving this paper. Early discussions with Steve Keckler, Arslan Zulfiqar, Jack Choquette and Olivier Giroux had a major influence on this work, for which we are very grateful. This work is supported by Nvidia through the GPU Center of Excellence program, the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), the Spanish Ministry of Science and Technology (TIN2015-65316-P) and by the Generalitat de Catalunya (grants 2014-SGR-1051 and 2014-SGR-1272). Nacho Navarro passed away before this paper was published. This work would have not been possible without his guidance, support, and dedication. A memory of him will always live in his students, colleagues and loved ones.

REFERENCES

[1] Andrew V. Adinetz and Dirk Pleiter. 2014. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures. http://on-demand.gputechconf.com/gtc/2014/presentations/S4271-halloc-high-throughput-dynamic-memory-allocator.pdf. (2014).

[2] AMD. 2016. Reference Guide: Graphics Core Next Architecture, Generation 3. (2016).

[3] DW Anderson, FJ Sparacio, and Robert M Tomasulo. 1967. The IBM System/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development* 11, 1 (1967), 8–24.

[4] A. W. Appel, J. R. Ellis, and K. Li. 1988. Real-time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. ACM, 11–20.

[5] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 96–107.

[6] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W. Hwu. 2012. Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 23–34.

[7] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux kernel*. " O'Reilly Media, Inc.".

[8] Roger A Bringmann, Scott A Mahlke, Richard E Hank, John C Gyllenhaal, and Wen-mei W Hwu. 1993. Speculative execution exception recovery using write-back suppression. In *Microarchitecture, 1993., Proceedings of the 26th Annual International Symposium on*. IEEE, 214–223.

[9] Werner Buchholz. 1986. The IBM System/370 vector architecture. *IBM systems journal* 25, 1 (1986), 51–62.

[10] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. 2006. Unbounded Page-based Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 347–358.

[11] Marc de Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 140–151.

[12] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, et al. 2002. Tarantula: a vector extension to the alpha architecture. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 281–292.

[13] Roger Espasa, Mateo Valero, and James E Smith. 1997. Out-of-order vector architectures. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 160–170.

[14] Robert Fitzgerald and Richard F Rashid. 1986. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems (TOCS)* 4, 2 (1986), 147–177.

[15] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. 2012. A hierarchical thread scheduler and register file for energy-efficient throughput processors. *ACM Transactions on Computer Systems (TOCS)* 30, 2 (2012), 8.

[16] Isaac Gelado, John H Kelm, Shane Ryoo, Steven S Lumetta, Nacho Navarro, and Wen-mei W Hwu. 2008. CUBA: an architecture for efficient CPU/co-processor data communication. In *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 299–308.

[17] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 347–358.

[18] Wen-mei W. Hwu and Yale N Patt. 1987. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th annual international symposium on Computer architecture*. ACM, 18–26.

[19] Thomas B Jablin, James A Jablin, Prakash Prabhu, Feng Liu, and David I August. 2012. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 165–174.

[20] Bruce L. Jacob and Trevor N. Mudge. 1998. A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 295–306.

[21] Feng Ji, Heshan Lin, and Xiaosong Ma. 2013. RSVM: a region-based software virtual memory for GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 269–278.

[22] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System.. In *USENIX Annual Technical Conference*. 401–412.

[23] Stephen W Keckler, Andrea Chang, WSLS Chatterjee, and William J Dally. 1999. Concurrent event handling through multithreading. *Computers, IEEE Transactions on* 48, 9 (1999), 903–916.

[24] Kenji Kitagawa, Satoru Tagaya, Yasuhiko Hagihara, and Yasushi Kanoh. 2003. A hardware overview of SX-6 and SX-7 supercomputer. *NEC research & development* 44, 1 (2003), 2–7.

[25] Christos Kozyrakis and David Patterson. 2003. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*. ACM, 399–409.

[26] George Kyriazis. 2012. Heterogeneous System Architecture: A Technical Review. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf. (2012).

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

[28] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2014. VAST: the illusion of a large memory space for GPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 443–454.

[29] Kai Li, J Naughton, and James Plank. 1990. Concurrent real-time checkpoint for parallel programs. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming*.

[30] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE* 28, 2 (2008), 39–55.

[31] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. 1992. Sentinel Scheduling for VLIW and Superscalar Processors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 238–247.

[32] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: exception support and speculative execution on GPUs. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 72–83.

[33] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. 1993. Register renaming and dynamic speculation: an alternative approach. In *26th annual international symposium on Microarchitecture*. IEEE, 202–213.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. *CACTI 6.0: A tool to model large caches*. Technical Report. Technical Report HPL-2009-85 HP Laboratories.

[35] NVIDIA. 2016. CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide. (2016).

[36] NVIDIA. 2016. NVIDIA Tesla P100 White Paper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. (2016).

[37] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 593–606.

[38] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on*

Architectural Support for Programming Languages and Operating Systems. ACM, 743–758.

[39] Jason Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 568–578.

[40] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A Variable Warp Size Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ACM, 489–501.

[41] Kevin W Rudd. 1997. *Efficient exception handling techniques for high-performance processor architectures*. Technical Report. Technical Report CSL-TR-97-732. Coordinated Science Laboratory, Stanford University.

[42] Richard M Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (1978), 63–72.

[43] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. 2009. Programming model for a heterogeneous x86 platform. In *ACM Sigplan Notices*, Vol. 44. ACM, 431–440.

[44] Michael Schulte. 2015. Floating-Point Arithmetic in AMD Processors. Presented at the 22nd IEEE Symposium on Computer Arithmetic, Lyon, France. (2015).

[45] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: a case for software address translation on GPUs. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*.

[46] James E. Smith and Andrew R. Pleszkun. 1985. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th annual International Symposium on Computer Architecture (ISCA '85)*. 36–44.

[47] Gurindar S Sohi et al. 1990. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE transactions on computers* 39, 3 (1990), 349–359.

[48] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu, and Wen-mei W. Hwu. 2012. *Parboil: a revised benchmark suite for scientific and commercial throughput computing*. Technical Report.

[49] Boleslaw K Szymanski. 1988. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd international conference on Supercomputing*. ACM, 621–626.

[50] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press, 193–204.

[51] James E Thornton. 1970. Design of a computer: the Control Data 6600. (1970).

[52] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[53] Hwa C. Torng and Martin Day. 1993. Interrupt handling for out-of-order execution processors. *IEEE Trans. Comput.* 42, 1 (1993), 122–127.

[54] Teruo Utsumi, Masayuki Ikeda, and Moriyuki Takamura. 1994. Architecture of the VPP500 parallel supercomputer. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 478–487.

[55] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171.

[56] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: Device memory management for GPGPU computing. In *The 2014 ACM international conference on Measurement and modeling of computer systems*. ACM, 533–545.

[57] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 358–369.

[58] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.

[59] Craig B Zilles, Joel S Emer, and Gurindar S Sohi. 1999. The use of multithreading for exception handling. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 219–229.