Triton-San: Toward Precise Debugging of Triton Kernels via LLVM Sanitizers

Lechen Yu[1], Tim Lu[2], Brandon Myers[1], Ofer Dekel[1]

[1]{yulechen, brandonmyers,oferd}@microsoft.com, [2] timthlu@gmail.com

1. Motivation

- Programmers may introduce programming errors in Triton kernels, which may result in
  - Incorrect result
  - Runtime error on GPU
- Multiple kinds of bugs may arise, including:
  - Buffer overflow
  - Use of uninitialized memory
  - Data race
- Silent and non-deterministic bugs could make debugging and reproducing difficult
- Triton doc website provides some guidance for debugging, but it lacks detailed examples and best practices for handling real-world bugs [1].

2. Case Study

Data Races in a Triton Kernel

```
 1 @triton.jit
 2 def kernel(output_ptr, n, BLOCK_SIZE):
 3   pid = tl.program_id(axis=0)
 4
 5   # Root cause of data race: incorrect block_start
 6   # Bug fix: block_start = pid * BLOCK_SIZE
 7   block_start = 0
 8
 9   offsets = block_start + tl.arange(0, BLOCK_SIZE)
10   mask = offsets < n
11   output = tl.full((BLOCK_SIZE, ), 1, dtype=tl.float16)
12
13   # Data races will occur between any two program ids
14   tl.store(output_ptr + offsets, output, mask=mask)
15
16 size = 256
17 output = torch.empty((size, )).to('gpu')
18 grid = lambda meta: (triton.cdiv(size, meta['BLOCK_SIZE']), )
19 kernel[grid](output, size, BLOCK_SIZE=2)
```

3. Triton-San [2]

- We developed a correctness tool tailored for Triton programs
  - currently capable of detecting buffer overflows and data races in Triton kernels.
- Execute the kernel on CPU through the hardware-agnostic middle-end *triton-shared* [3]
  - Put the focus on programming errors not compiler errors
  - Compile Triton kernels through the built-in MLIR->LLVM IR->Assembly lowering
  - Doesn't require a GPU/NPU backend when debugging
- Parallelize kernel execution on CPU via LLVM OpenMP runtime [4]
- Leverage LLVM sanitizers [5] to ensure reliability and maintainability
  - LLVM sanitizers (from the compiler-rt subproject) have been extensively optimized to detect bugs with high efficiency
  - AddressSanitizer (ASan): catches buffer overflows
  - ThreadSanitizer (TSan): catches data races

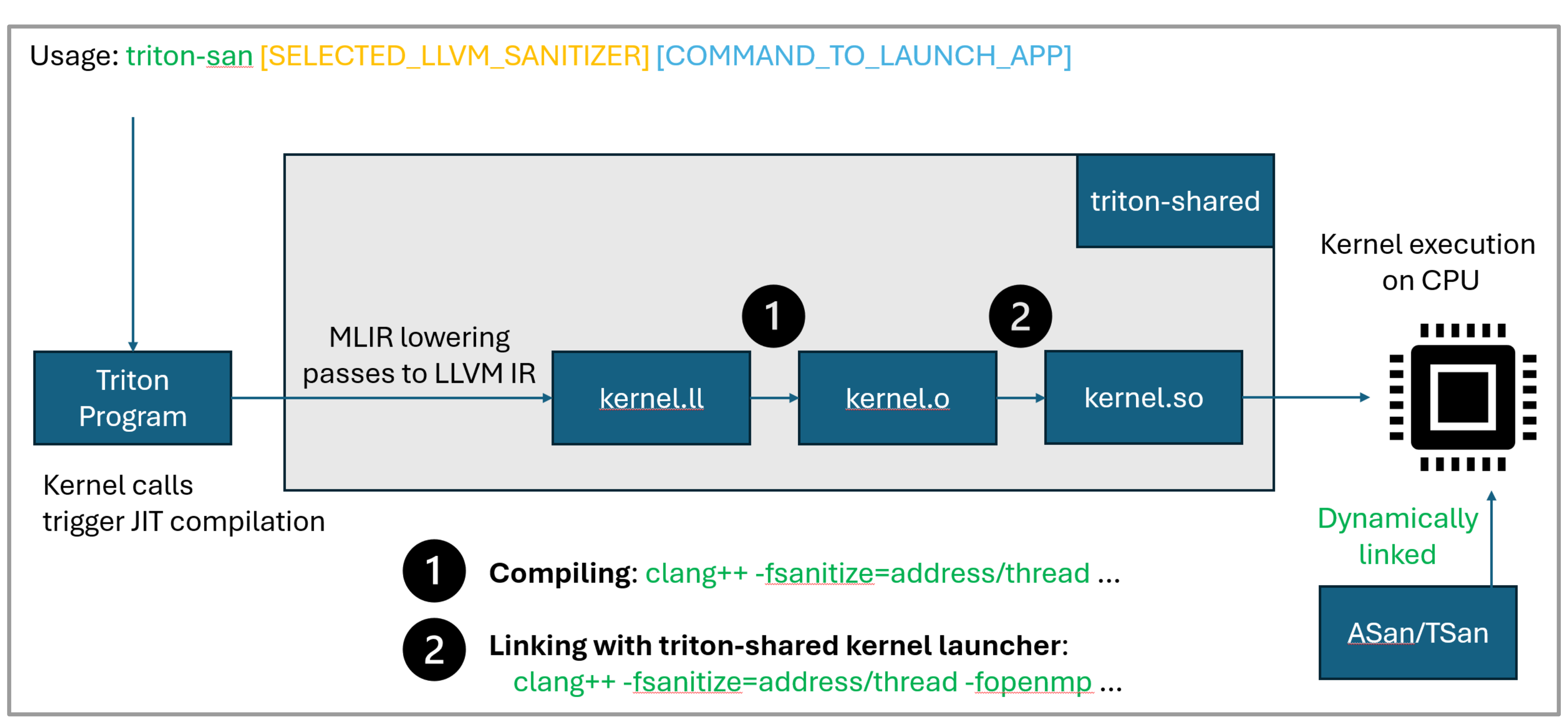
4. Triton-San Bug Report

```
WARNING: ThreadSanitizer: data race (pid=3823476)
  Write of size 8 at 0x7260004bd800 by thread T44:
    #0 __tsan_memcpy tsan_interceptors_memintrinsics.cpp:27:3
    #1 kernel data-race.py:14:35
    #2 _launch(int, int, int, void*, int, _object*) (.omp_outlined_debug__)
/tmp/tmpb3gw_jsv/main.cxx:26:11
    #3 _launch(int, int, int, void*, int, _object*) (.omp_outlined)
/tmp/tmpb3gw_jsv/main.cxx:20:5
    #4 __kmp_invoke_microtask <null> (libomp.so+0xc3d98)
    #5 _launch(int, int, int, void*, int, _object*) /tmp/tmpb3gw_jsv/main.cxx:20:5
(__triton_shared_ref_cpu_kernel_launcher.so+0x764a)

  Previous write of size 8 at 0x7260004bd800 by main thread:
    #0 __tsan_memcpy tsan_interceptors_memintrinsics.cpp:27:3
    #1 kernel data-race.py:14:35
    #2 _launch(int, int, int, void*, int, _object*) (.omp_outlined_debug__)
/tmp/tmpb3gw_jsv/main.cxx:26:11
    #3 _launch(int, int, int, void*, int, _object*) (.omp_outlined)
/tmp/tmpb3gw_jsv/main.cxx:20:5
    #4 __kmp_invoke_microtask <null> (libomp.so+0xc3d98)
    #5 _launch(int, int, int, void*, int, _object*) /tmp/tmpb3gw_jsv/main.cxx:20:5
(__triton_shared_ref_cpu_kernel_launcher.so+0x764a)
    #6 <null> <null> (python3.12+0x581a8f)
```

5. System Overview of Triton-San

On-the-fly Dynamic Analysis for Triton Kernels



6. Effectiveness of Triton-San

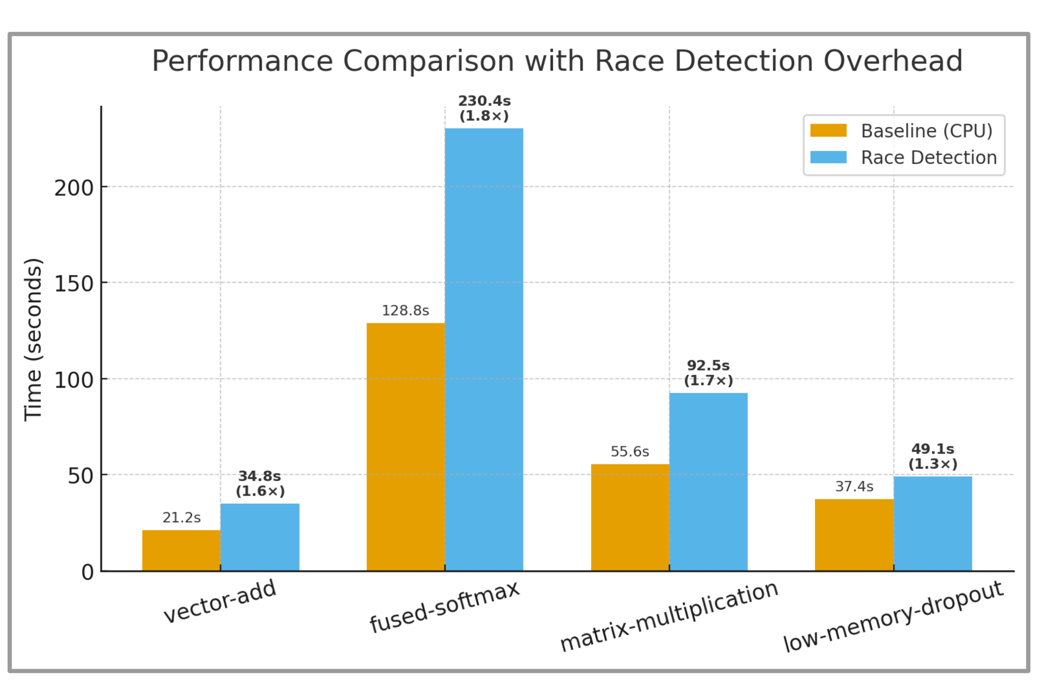
- Developed micro-benchmarks to test the effectiveness of correctness tools when tackling Triton kernels
- Each benchmark contains one injected bug

Error Type	Compute Sanitizer [6]	Triton-San
Buffer overflow	❌	✅
Data race	❌	✅

- Triton-San detected all injected bugs with no false alarms
- Compute Sanitizer showed several limitations
  - No data races detected (none occur in shared memory)
  - Detected buffer overflows only on very large tensors

7. Evaluations of Runtime Overhead

- Performed a preliminary performance evaluation
- Used four benchmarks from Triton tutorials
  - vector-add
  - fused-softmax
  - matrix-multiplication
  - low-memory-dropout
- Launched Triton kernels with 16 CPU threads as the baseline version



8. Other Correctness Tools for Triton

- For Triton kernels running on Nvidia GPUs, using correctness tools from the CUDA toolkit (i.e., *Nvidia Compute Sanitizer* [6]) can serve as an alternative for debugging and validation.
  - Racecheck: targeting only shared-memory data races
  - Memcheck: targeting memory issues (e.g, buffer overflow, memory leak)
  - Initcheck: targeting uninitialized memory accesses
- Our evaluations show that Compute Sanitizer is not very effective at identifying Triton bugs
- ConSan, a new tool for Triton, is under active development

[1]. Debugging Triton. https://triton-lang.org/main/programming-guide/chapter-3/debugging.html
[2]. Triton-San. https://github.com/microsoft/triton-shared/tree/main/triton-san
[3]. Triton-Shared. https://github.com/microsoft/triton-shared
[4]. LLVM OpenMP Support. https://clang.llvm.org/docs/OpenMPSupport.html
[5]. LLVM Compiler-RT. https://compiler-rt.llvm.org
[6]. Nvidia Compute Sanitizer. https://developer.nvidia.com/compute-sanitizer

Microsoft Triton