Microsoft

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

Lechen Yu, **Tim Lu**, Brandon Myers, Ofer Dekel

Motivation

- Python-based GPU programming has seen rapid adoption in HPC and AI
  - E.g., Triton, cuTile, Helion, Numba, PyOMP
  - Gentle learning curve
  - Seamless integration with other Python libraries (e.g., PyTorch)
- Incorrect usage of provided constructs may still lead to concurrency bugs and memory issues
- Currently, there are few tools available to help programmers diagnose root causes in Python programming

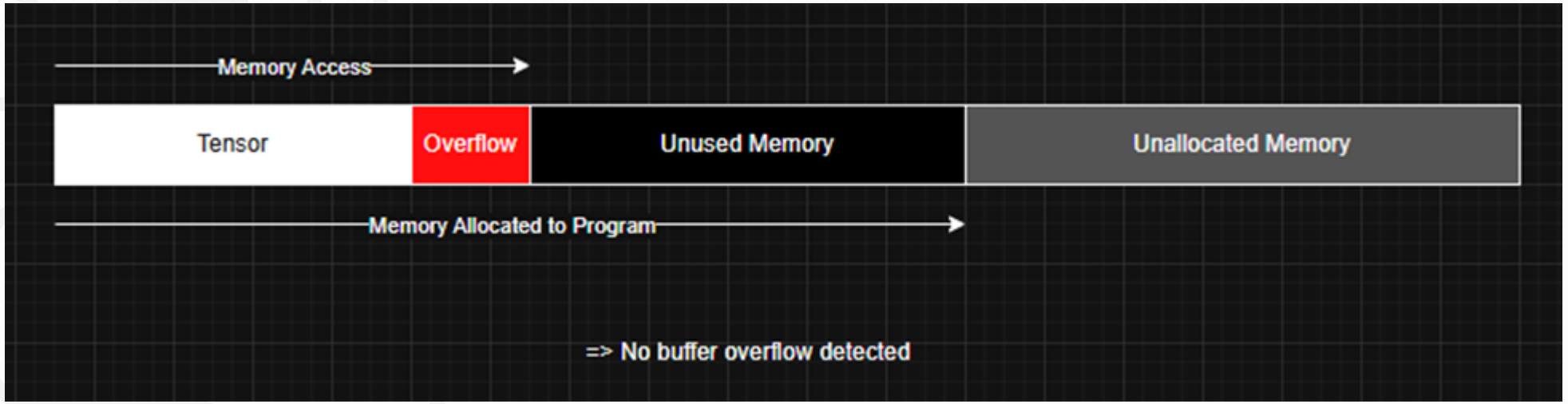
Challenging Programming Errors

- *Buffer overflow*: reads/writes outside of the bounds of allocated memory
- *Use of uninitialized memory*: usage before initialization
- *Data races*: two threads access memory without synchronization and at least one access is a write

- Silent and non-deterministic behaviors are hard to detect

Example of Silent Buffer Overflow

- Could be silent error due to the over-subscription of GPU memory
- Incurred due to memory planning in the Triton compiler



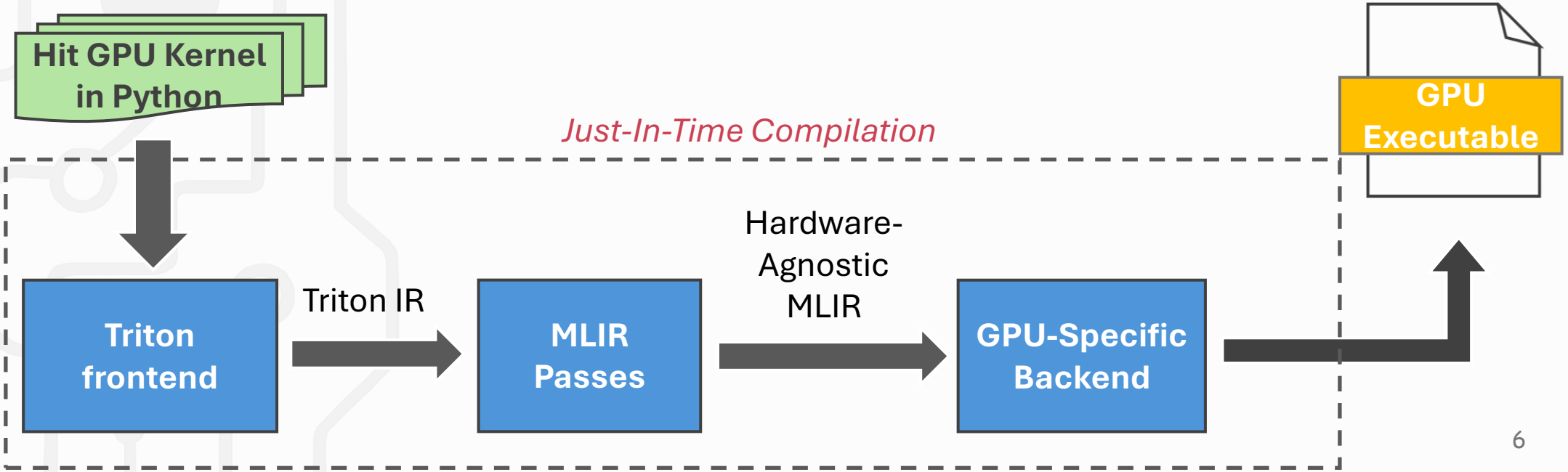
Our Work: Triton-San

- A dynamic analysis tool to pinpoint such programming errors
- Leverage LLVM Sanitizers for reliability and maintainability
- Offload the kernel execution to CPU when examining a Triton program
  - Utilize the built-in MLIR->LLVM IR-> Assembly pipeline
  - Parallelize the kernel execution on CPU through the LLVM OpenMP runtime

Background – Triton

- Triton: open source, Python-based programming language + just-in-time compiler for GPU kernels

- How Does Triton Work:



Background – Triton Program

1. Python interpreter loop

```
1
2
3
4
5
6
7
8
9
10
11
12
13  size = 256
14  output = torch.empty((size, )).to('gpu')
15  grid = lambda meta: (triton.cdiv(size, meta['BLOCK_SIZE']), )
16
```

Background – Triton Program

1. Python interpreter loop
2. Kernel call is reached

```
1
2
3
4
5
6
7
8
9
10
11
12
13  size = 256
14  output = torch.empty((size, )).to('gpu')
15  grid = lambda meta: (triton.cdiv(size, meta['BLOCK_SIZE']), )
16  kernel[grid](output, size, BLOCK_SIZE=2)
```

Background – Triton Program

1. Python interpreter loop
2. Kernel call is reached
3. Kernel Just-In-Time (JIT) compilation

```
1  @triton.jit
2  def kernel(output_ptr, n, BLOCK_SIZE):
3
4
5
6
7
8
9
10
11
12
13 size = 256
14 output = torch.empty((size, )).to('gpu')
15 grid = lambda meta: (triton.cdiv(size, meta['BLOCK_SIZE']), )
16 kernel[grid](output, size, BLOCK_SIZE=2)
```

Background – Triton Program

1. Python interpreter loop
2. Kernel call is reached
3. Kernel Just-In-Time (JIT) compilation
4. Kernel execution
   - Single Program, Multiple Data (SPMD)
   - Uses specified Triton grid

```
1  @triton.jit
2  def kernel(output_ptr, n, BLOCK_SIZE):
3      pid = tl.program_id(axis=0)
4
5      block_start = 0
6
7      offsets = block_start + tl.arange(0, BLOCK_SIZE)
8      mask = offsets < n
9      output = tl.full((BLOCK_SIZE, ), 1, dtype=tl.float16)
10
11     tl.store(output_ptr + offsets, output, mask=mask)
12
13 size = 256
14 output = torch.empty((size, )).to('gpu')
15 grid = lambda meta: (triton.cdiv(size, meta['BLOCK_SIZE']), )
16 kernel[grid](output, size, BLOCK_SIZE=2)
```

Background – Triton Program

1. Python interpreter loop
2. Kernel call is reached
3. Kernel Just-In-Time (JIT) compilation
4. Kernel execution
   - Single Program, Multiple Data (SPMD)
   - Uses specified Triton grid
5. Control returned to Python interpreter

```
1  @triton.jit
2  def kernel(output_ptr, n, BLOCK_SIZE):
3    pid = tl.program_id(axis=0)
4
5    block_start = 0
6
7    offsets = block_start + tl.arange(0, BLOCK_SIZE)
8    mask = offsets < n
9    output = tl.full((BLOCK_SIZE, ), 1, dtype=tl.float16)
10
11   tl.store(output_ptr + offsets, output, mask=mask)
12
13 size = 256
14 output = torch.empty((size, )).to('gpu')
15 grid = lambda meta: (triton.cdiv(size, meta['BLOCK_SIZE']), )
16 kernel[grid](output, size, BLOCK_SIZE=2)
```

Python execution continues ↓

Data Race in Triton Program

```
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)
```

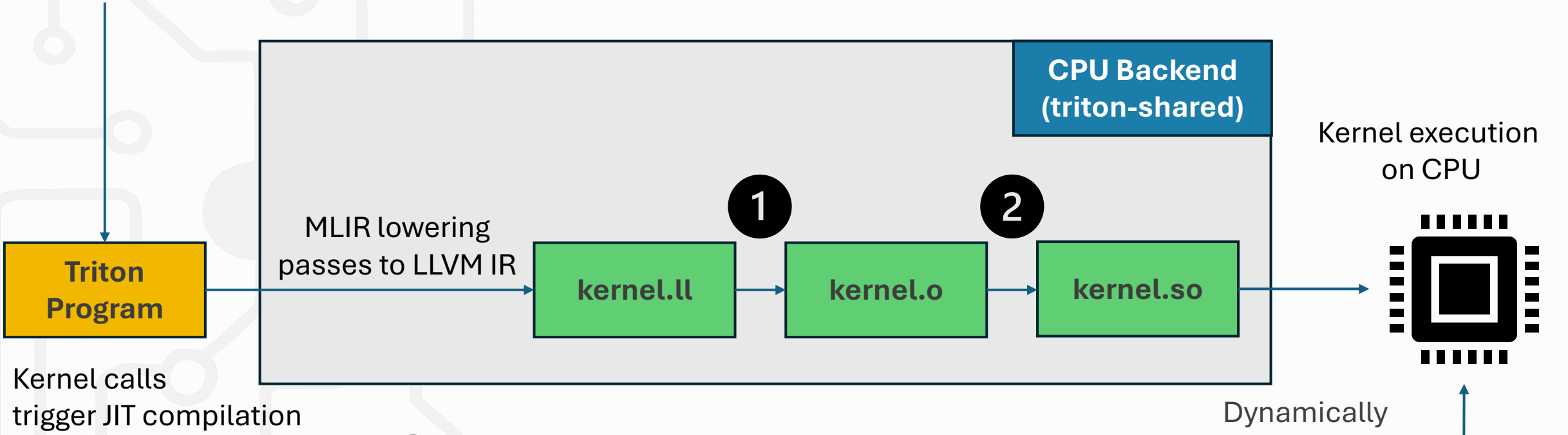
Buffer Overflow in Triton Program

```
1  @triton.jit
2  def kernel(output_ptr, n, BLOCK_SIZE):
3    pid = tl.program_id(axis=0)
4
5    block_start = pid * BLOCK_SIZE
6    offsets = block_start + tl.arange(0, BLOCK_SIZE)
7
8    # Root cause: mask is erroneously set to the whole tensor
9    mask = tl.full((BLOCK_SIZE, ), 1, dtype=tl.int1)
10
11   output = tl.full((BLOCK_SIZE, ), 1, dtype=tl.float16)
12
13   # Out-of-bound access (buffer overflow) due to mask
14   tl.store(output_ptr + offsets, output, mask=mask)
15
16 size = 3
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)
```

Triton-San

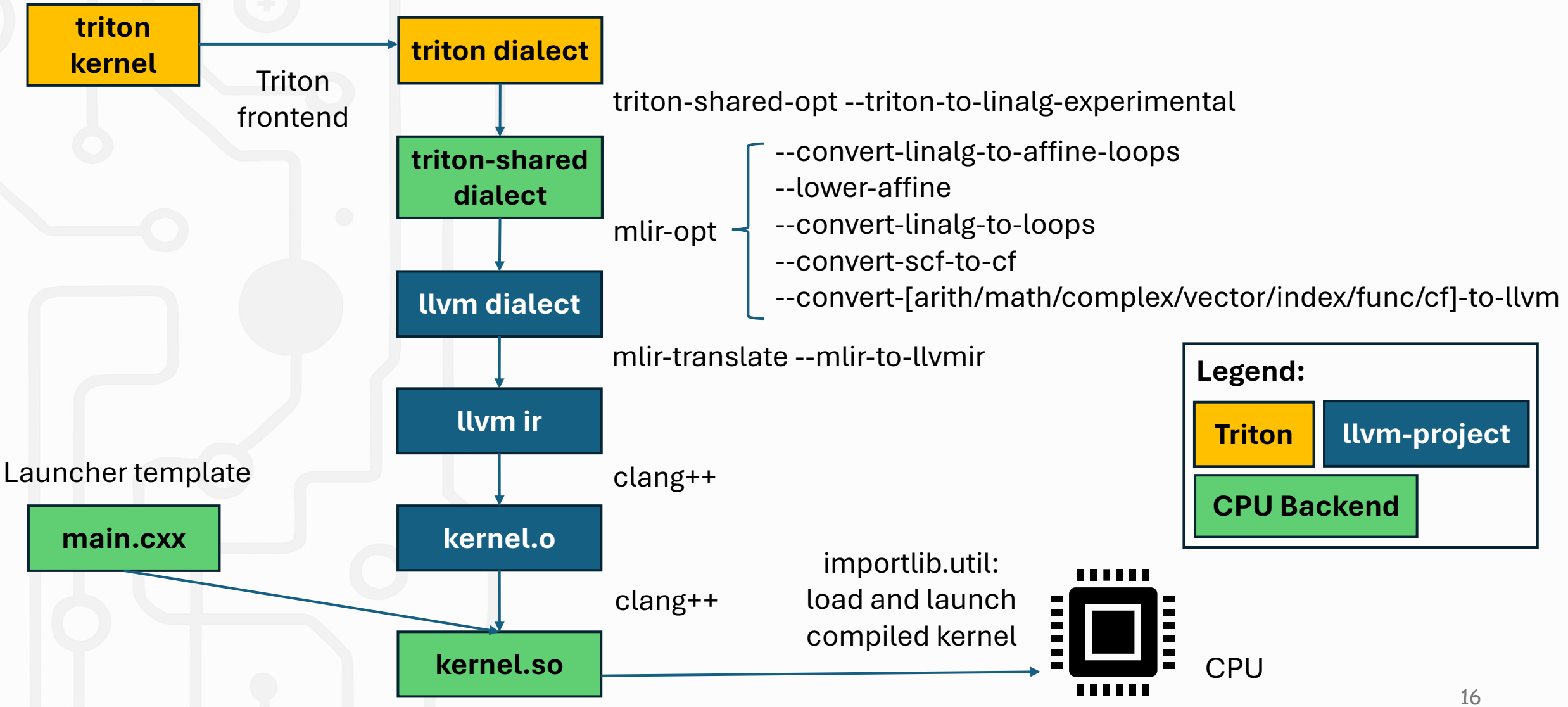
Triton-San's Workflow

Usage: triton-san [SELECTED_LLVM_SANITIZER] [COMMAND_TO_LAUNCH_APP]



1. **Compiling**: clang++ -fsanitize=address/thread ...
2. **Linking with triton-shared kernel launcher**:
   clang++ -fsanitize=address/thread -fopenmp ...

Triton-San: Offload Kernel to CPU



Triton-San: Parallelize Triton Grid Loop

- Problem: triton-shared translates kernels into a sequential CPU program
  - Bad performance
  - No data races can occur and go undetected
- Solution: parallelize kernel execution with OpenMP
  - Each grid instance is assigned its own OpenMP worker thread
  - ThreadSanitizer officially supports OpenMP in its LLVM extension Archer

```
{"#pragma omp parallel for collapse(3)" if _get_sanitizer_type() == "tsan" else ""}
for(int x = 0; x < gridX; x++) {{
  for(int y = 0; y < gridY; y++) {{
    for(int z = 0; z < gridZ; z++) {{
      // Use some random type "char" here.
      {' '.join(f'StridedMemRefType<char, 0> ptr_arg{i} = {{static_cast<char *>(arg{i}),
      {kernel_name}({kernel_parameters}
                    gridX, gridY, gridZ, x, y, z);
```

Triton-San: Fix Missing Debug Info

- Problem: debug information needs to be available during runtime to be reported
- Solution: MLIR passes need to add + retain relevant debug info

```
==================
WARNING: ThreadSanitizer: data race (pid=1685624)
  Write of size 8 at 0x7260004e7000 by thread T48:
    #0 __tsan_memcpy /home/yulechen/Repository/triton/release/llvm/llvm-project
/compiler-rt/lib/tsan/rtl/tsan_interceptors_memintrinsics.cpp:27:3 (libclang_rt
.tsan.so+0x6d2de) (BuildId: f597c15bf4e434502c65fb6fa56e15a1099aefde)
    #1 kernel /home/yulechen/Repository/triton/release/triton-san/./examples/da
ta-race.py:28:35 (__triton_shared_ref_cpu_kernel_launcher.so+0xcdaa)
    #2 _launch(int, int, int, void*, int, _object*) (.omp_outlined_debug__) /tm
p/tmp_utokc8q/main.cxx:26:11 (__triton_shared_ref_cpu_kernel_launcher.so+0x7bc9
)
    #3 _launch(int, int, int, void*, int, _object*) (.omp_outlined) /tmp/tmp_ut
okc8q/main.cxx:20:5 (__triton_shared_ref_cpu_kernel_launcher.so+0x7ca1)
    #4 __kmp_invoke_microtask <null> (libomp.so+0xc41d8) (BuildId: 9506660c39cd
03f9cee9efd3363cd39cc968e672)
    #5 _launch(int, int, int, void*, int, _object*) /tmp/tmp_utokc8q/main.cxx:2
0:5 (__triton_shared_ref_cpu_kernel_launcher.so+0x764a)
```

```
#di_file = #llvm.di_file<"buffer-overflow-write.py" in "/workspace/micro-benchma
#di_subroutine_type = #llvm.di_subroutine_type<callingConvention = DW_CC_normal>
#loc = loc("/workspace/micro-benchmarks/buffer-overflow-write.py":9:0)
#di_compile_unit = #llvm.di_compile_unit<id = distinct[0]<>, sourceLanguage = DW
#di_subprogram = #llvm.di_subprogram<id = distinct[1]<>, compileUnit = #di_compi
#loc5 = loc(fused<#di_subprogram>[#loc])
module {
  func.func @kernel(%arg0: memref<*xf32> {tt.divisibility = 16 : i32} loc("/works
    %cst = arith.constant 1.000000e+00 : f32 loc(#loc1)
```

Triton-San: Link/Load Sanitizers

- Sanitizer is preloaded before the Triton program starts its execution
  - LD_PRELOAD=libclang_rt.asan/tsan.so python triton_program.py

- Allows sanitizer to initialize before the program runs
  - Allocate shadow memory to track execution metadata
  - Register hook callbacks to intercept OS-level APIs
  - Set up suppression lists to exclude non-Triton modules from instrumentation (e.g. PyTorch, NumPy)

Triton-San: Filter and Suppression

- Suppressing false positives: use built-in mechanisms
- Driver script runs with:

```
TSAN_OPTIONS="ignore_noninstrumented_modules=0:report_mutex_bugs=0:suppressions=${SUPPRESSION_FILE}"
```

- Suppression file: filters out non-kernel modules

```
called_from_lib:libomp.so
called_from_lib:libtorch_python.so
called_from_lib:libtorch_cpu.so
called_from_lib:libtorch_cuda.so"
```

Evaluation

Evaluation Method

- 2 evaluation datasets:
  - Micro-benchmarks: minimal Triton kernel designed to expose a specific class of bug
  - Realistic cases + injected bugs: Triton kernels from official Triton tutorial and Triton Bench
- 2 sets of evaluation experiments:
  - Precision: successful detection of bugs of interest
  - Performance: memory + time overhead
- Environment:
  - 64-core AMD EPYC 7763 processor + 128 GB of memory
  - Ubuntu 24.04
  - Triton commit 65d9862f5a9029827a7ae04439f07d7e7eadf859

Precision

- Successfully identified all injected issues in the micro-benchmark suite without false positives

```
22
23      # write output to the output_ptr
24      # this write will be a buffer overflow
25      tl.store(output_ptr + offsets, input_, mask=mask)
26
```

```
(venv) t-timlu@TDC2021955691:/workspace/micro-benchmarks$ LD_PRELOAD="/workspace/triton_shared/triton/li
nux-gnu/libclang_rt.asan.so" python buffer-overflow-write.py
tensor([-0.3725, -0.3725])
warning: overriding the module target triple with x86_64-unknown-linux-gnu [-Woverride-module]
1 warning generated.
=================================================================
==1578767==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7c0cb9de0288 at pc 0x7f7cbb547775
WRITE of size 4194304 at 0x7c0cb9de0288 thread T0
    #0 0x7f7cbb547774 in __asan_memcpy /workspace/triton_shared/triton/llvm-project/compiler-rt/lib/asan
    #1 0x7f7cbaeb9371 in kernel /workspace/micro-benchmarks/buffer-overflow-write.py:25:35
    #2 0x7f7cbaeb2e03 in _launch(int, int, int, void*, int, _object*) /tmp/tmp_p4vk1_r/main.cxx:23:11
    #3 0x7f7cbaeb264e in launch(_object*, _object*) /tmp/tmp_p4vk1_r/main.cxx:103:3
    #4 0x0000005820af  (/usr/bin/python3.12+0x5820af) (BuildId: 37451b37c71cb46f8ccb27cb3cdbb7aa004b9987
    #5 0x00000054b30b in PyObject_Call (/usr/bin/python3.12+0x54b30b) (BuildId: 37451b37c71cb46f8ccb27cb
```

Comparison With Compute Sanitizer

- For Triton kernels running on Nvidia GPUs, using *Nvidia Compute Sanitizer* 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

	compute-sanitizer	Triton-San
Buffer overflow	Can only detect large overflows	Can detect overflows of any size
Use of uninitialized memory	Cannot detect uses of uninitialized memory in some cases	To be supported in the future
Data races	Cannot detect data races	Can detect data races

Nvidia Compute Sanitizer. https://developer.nvidia.com/compute-sanitizer

Performance

- Similar overhead to existing applications

Sanitizer	Dataset	Time Usage	Memory Usage
No sanitizers – 1 thread	Triton tutorials	1x	1x
	Triton bench (aiter)	1x	1x
AddressSanitizer – 1 thread	Triton tutorials	1.66x	1.57x
	Triton bench (aiter)	1.60x	1.36x
Sanitizer	**Dataset**	**Time Usage**	**Memory Usage**
No sanitizers – 16 threads	Triton tutorials	1x	1x
	Triton bench (aiter)	1x	1x
ThreadSanitizer – 16 threads	Triton tutorials	2.84x	2.98x
	Triton bench (aiter)	2.32x	2.76x

Triton-San Setup and Usage

Installing Triton-San

- Triton-San has been integrated into triton-shared as an optional feature

- Setup
  - 1) clone triton-shared
  - 2) run ./triton-san/build.sh

- The build script
  - Installs compatible versions of LLVM, Triton, and triton-shared
  - Generates the driver script “triton-san” for simple invocation

Using Triton-San

1. In the Triton program, add required imports to specify the use of the triton-shared CPU backend:

```
from triton.backends.triton_shared.driver import CPUDriver
triton.runtime.driver.set_active(CPUDriver())
```

2. Set all desired GPU tensors to CPU

```
# output = torch.empty((size, )).to("gpu")
output = torch.empty((size, )).to("cpu")
```

3. Run the driver script ./triton-san/triton-san with the target Triton program and corresponding inputs

```
# Run triton-san/triton-san to view usage instructions.

Usage: triton-san <sanitizer type> <original command used to launch the triton program...>.
<sanitizer type>:
  "asan": to detect buffer overflows
  "tsan": to detect data races

Example: triton-san asan python ./my_triton_program.py
```

Summary

- New trend in Python-based GPU programming, e.g. Triton
- Debugging in Triton is difficult – limited tools and resources
- Triton-San: combines LLVM sanitizers with Triton's CPU backend to precisely detect elusive issues
- Result:
  - Enhance debugging toolkit for Triton devs
  - Extend LLVM sanitizers to python-based languages

Microsoft

Thank You

Triton-San

Usage: triton-san [SELECTED_LLVM_SANITIZER] [COMMAND_TO_LAUNCH_APP]