The Proton Dialect: An MLIR Dialect For AI Compiler GPU Kernel Profiling

Keren Zhou (kzhou6@gmu.edu) Corbin Robeck (corbin.robeck@amd.com)
Yuanwei Fang (fywkevin@meta.com)

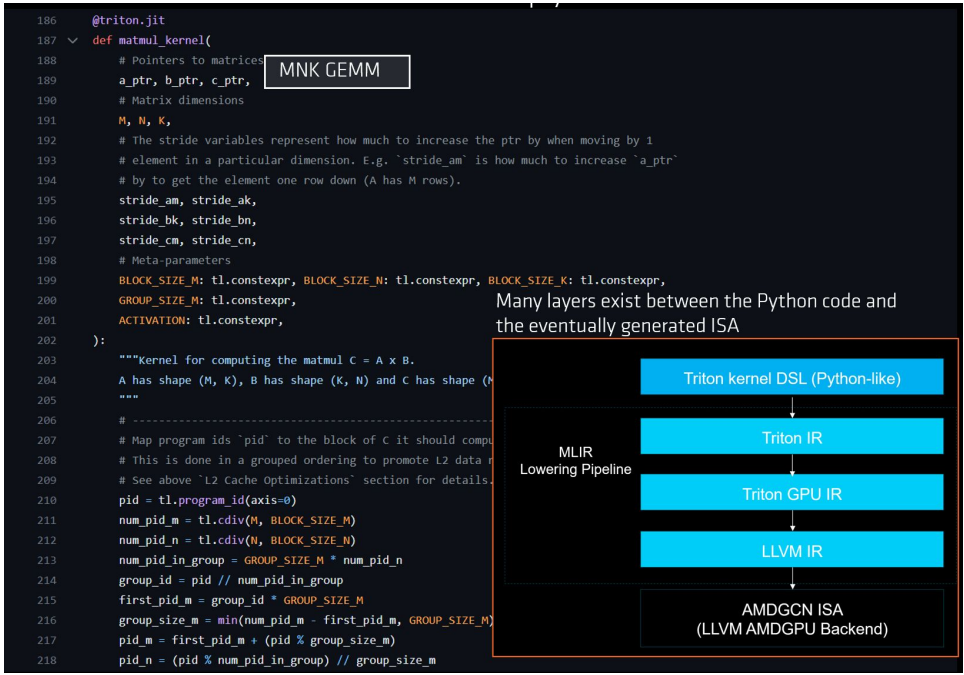
Background and Motivation

- MLIR based AI compilers have become popular to bridge the gap between high-level machine learning framework operators (GEMMs, softmax, etc.) and low-level, target-specific machine code.
  - Sophisticated compiler passes are required however to map high level MLIR operators to low level LLVM and target specific code
- There has also been a recent push to incorporate more non-traditional compiler operations into MLIR dialects (e.g. MPI Dialect for comms)
  - Defining operations directly in the language allows operation specific lowering
  - This becomes important when MLIR level passes are used to reordering instructions above the LLVM IR
    - Software pipelining, shared memory pre-fetching, etc.
- Recent advancements in Large Language Model (LLM) architecture and optimization (e.g. DeepSeek) have shown that the ability to reorder instructions on the IR/Assembly level can have outsized performance impacts

Triton MLIR Based ML Compiler

Triton is popular MLIR based framework for implementing high-performance AI operators (GEMMs, Flash Attention, etc) on GPUs (AMD and Nvidia)

- Complex MLIR passes implement operator specific transforms to achieve high performance without requiring users to be GPU architecture experts
- This has shifted performance many optimizations from kernel writers to compiler and framework developers



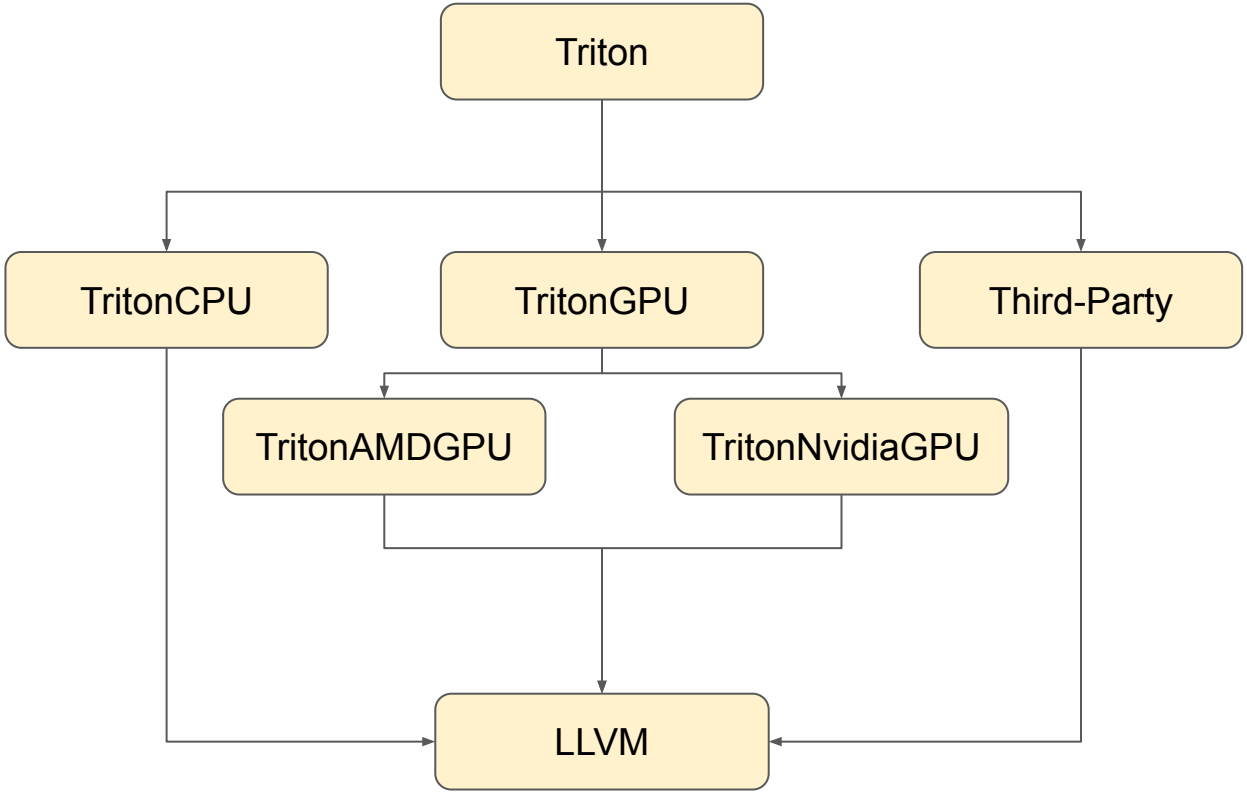
Optimizing Code on Modern GPUs

- The latest generation of data center GPUs (H100, MI300) make use of specialized functional units specifically for acceleration of matrix multiplication (Tensor/Matrix cores)
- To make optimal use of these features requires complex software abstractions
  - Warp specialization (Nvidia)
  - Wave priority and scheduling (AMD)
  - Loop pipelining and instruction reordering
- Optimizing these SW implementations requires fine-grained, intra-kernel tracing capabilities that do not break the complex optimization passes

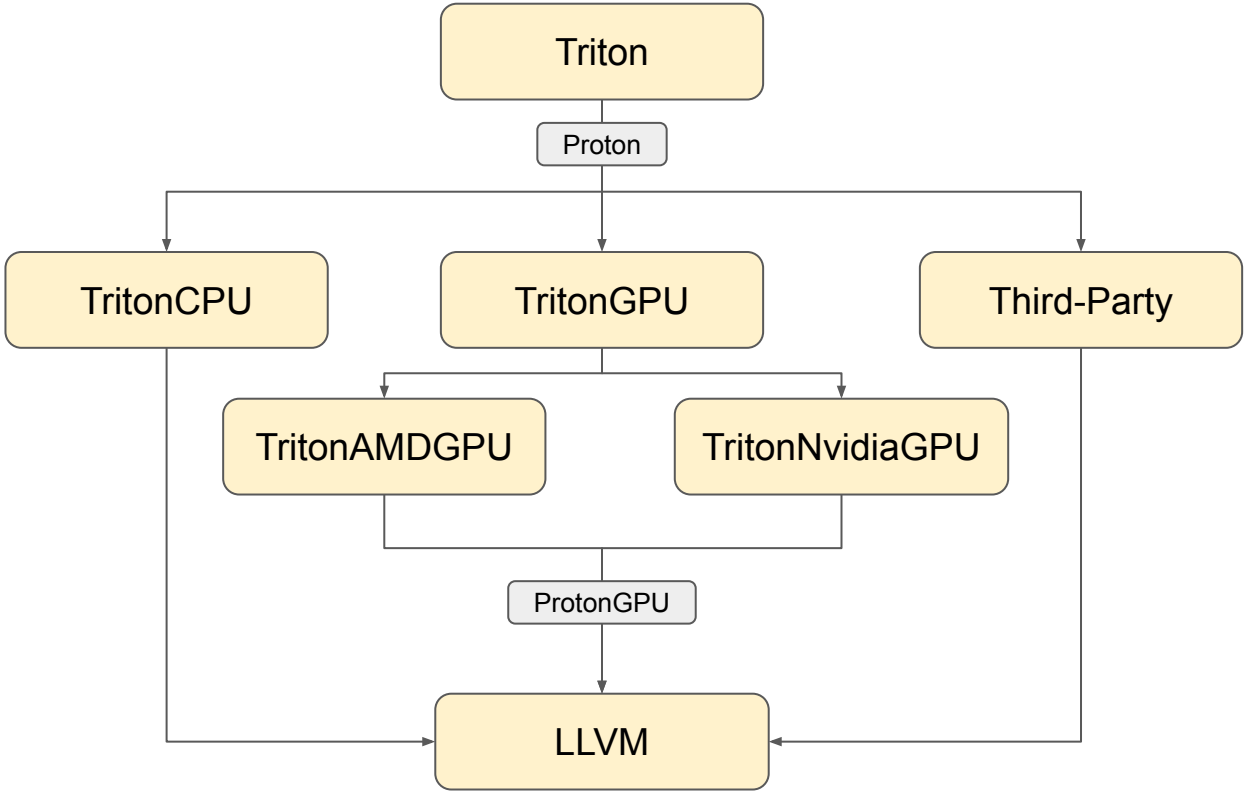
Custom Instrumentation: Beyond CUPTI & RocTracer

- Limitations of existing backends
  - CUPTI and RocTracer are powerful but may not fully address our needs
- Why custom instrumentation?
  - Cross-platform support: One engine for multiple GPUs/accelerators
  - Reusable utilities: Simplify development across kernels
  - Extended metrics: Capture data unavailable through vendor tools
- Examples
  - Memory heat map generation to visualize performance bottlenecks
  - Tailored instrumentation for asynchronous matrix multiplication instructions

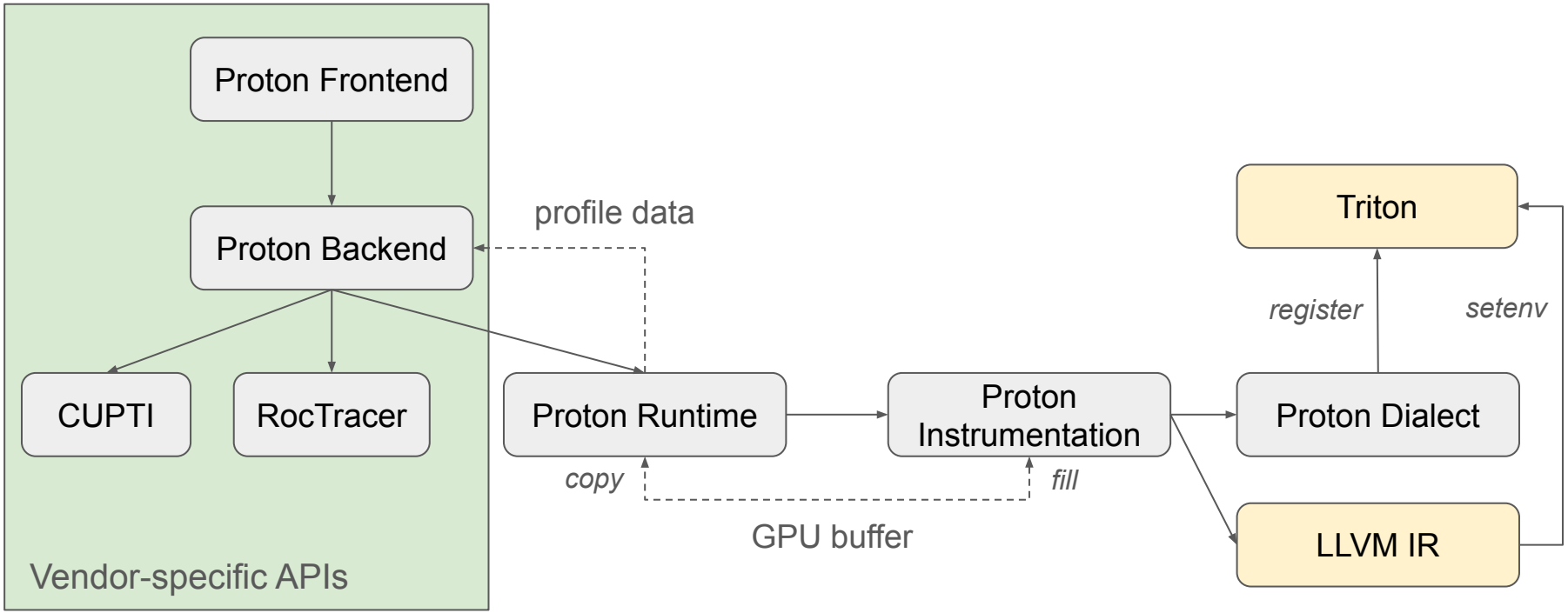
Dialect Overview



Proton Dialects



Proton Runtime



Usage

- Python API
  - Instrument Triton Python code
- Proton dialect instrumentation
  - Generic for any backend
  - Compiler engineers can specify recording start/end scopes
- ProtonGPU dialect instrumentation
  - Generated by the instrumentation backend
    - Measuring specific hardware/software metrics

Proton Language

The Proton Dialect Language augments the existing Triton Language

```python
import torch
import pytest
import pathlib

import triton
import triton.language as tl

@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)
torch.manual_seed(0)
size = 2**12
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
output = torch.empty_like(x)
n_elements = output.numel()
grid = (1, 1, 1)
add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
```

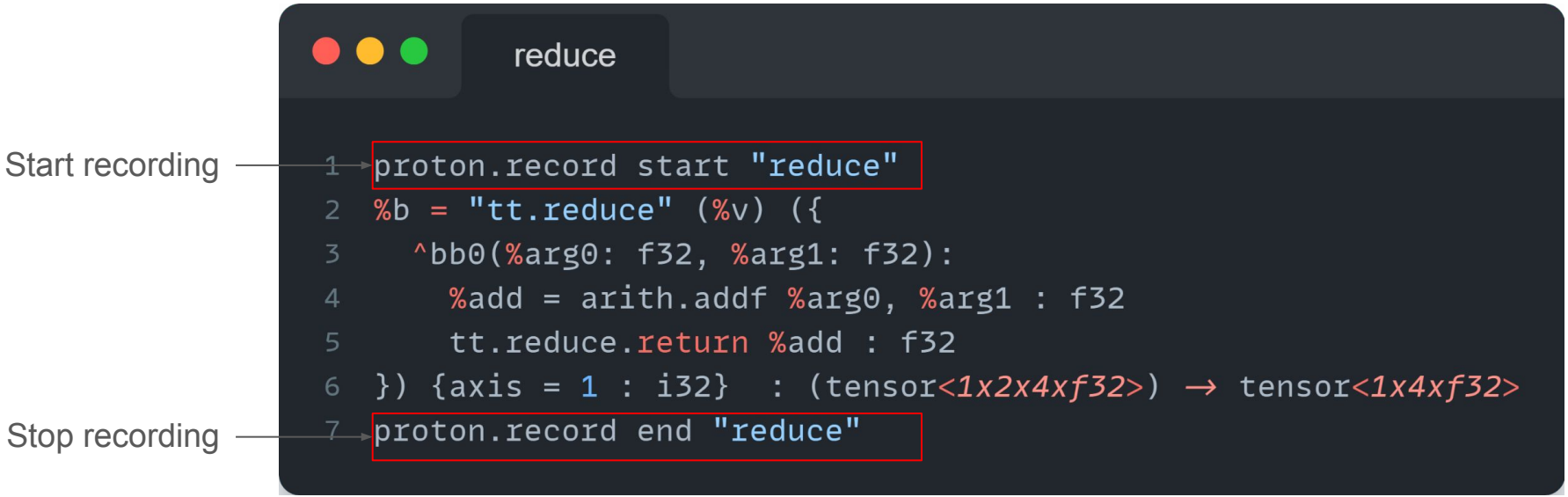
```python
import torch
import pytest
import pathlib

import triton
import triton.language as tl
import triton.profiler.language as pl

@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    pl.record(True, 0)
    y = tl.load(y_ptr + offsets, mask=mask)
    pl.record(False, 0)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)
torch.manual_seed(0)
size = 2**12
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
output = torch.empty_like(x)
n_elements = output.numel()
grid = (1, 1, 1)
add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
```

Python API

- `proton.start(backend="instrumentation", mode="...")`
  - Patches all Triton functions with the given mode
  - Each mode specifies
    - What metrics to profile
    - Sampling modes
    - Collection granularity
  - Example: `mma_cycle::[warpgroup::circular::all]`
    - `[warpgroup::circular::all]` is optional

Proton Dialect MLIR Level Instrumentation

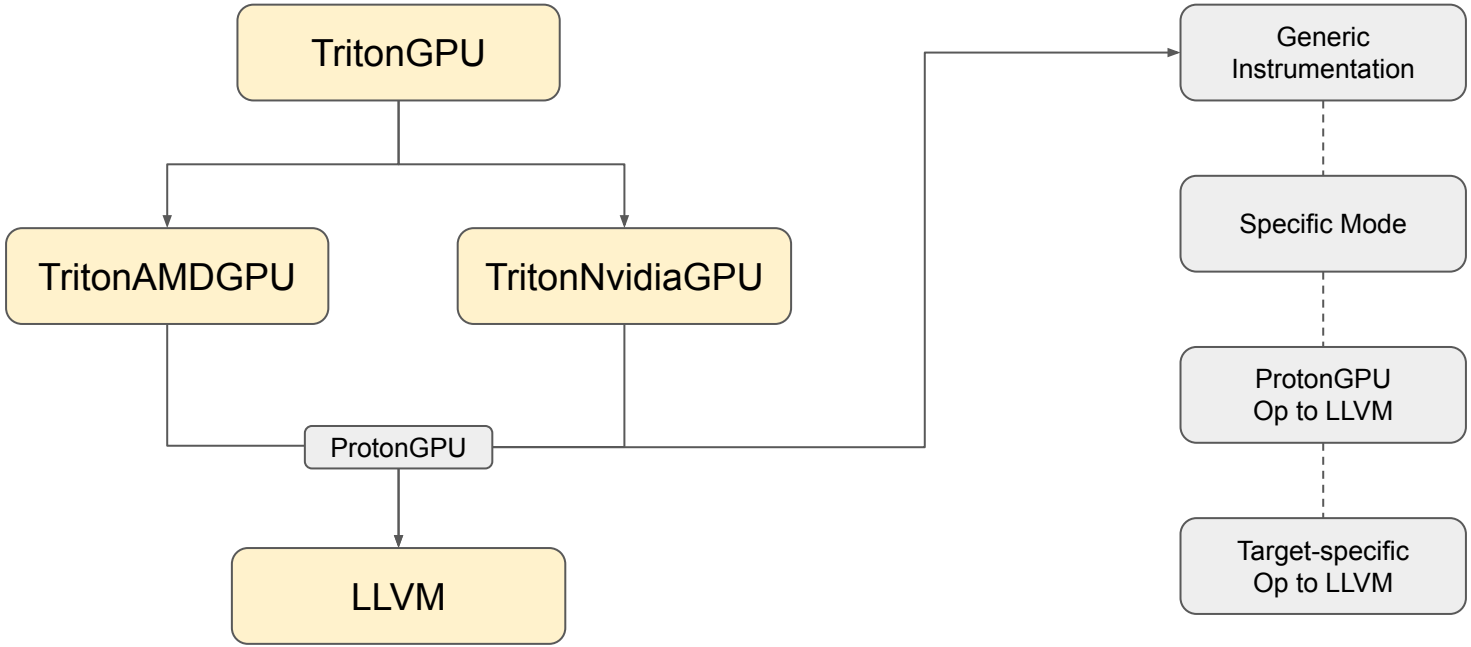
```
proton.record start/end “scope_name”
```



ProtonGPU Dialect Instrumentation

- `proton_gpu.global_scratch_alloc`
  - Obtain a pointer from the global profile data
- `proton_gpu.init_buffer_index`
  - Initial an index for recording records in the local buffer
- `proton_gpu.read_counter`
  - Read a performance counter value at this point
- `proton_gpu.circular_store`
  - Store a record in the local buffer and increase the local index
- `proton_gpu.finalize`
  - Copy the local buffer to the global profile data

ProtonGPU to LLVM Lowering



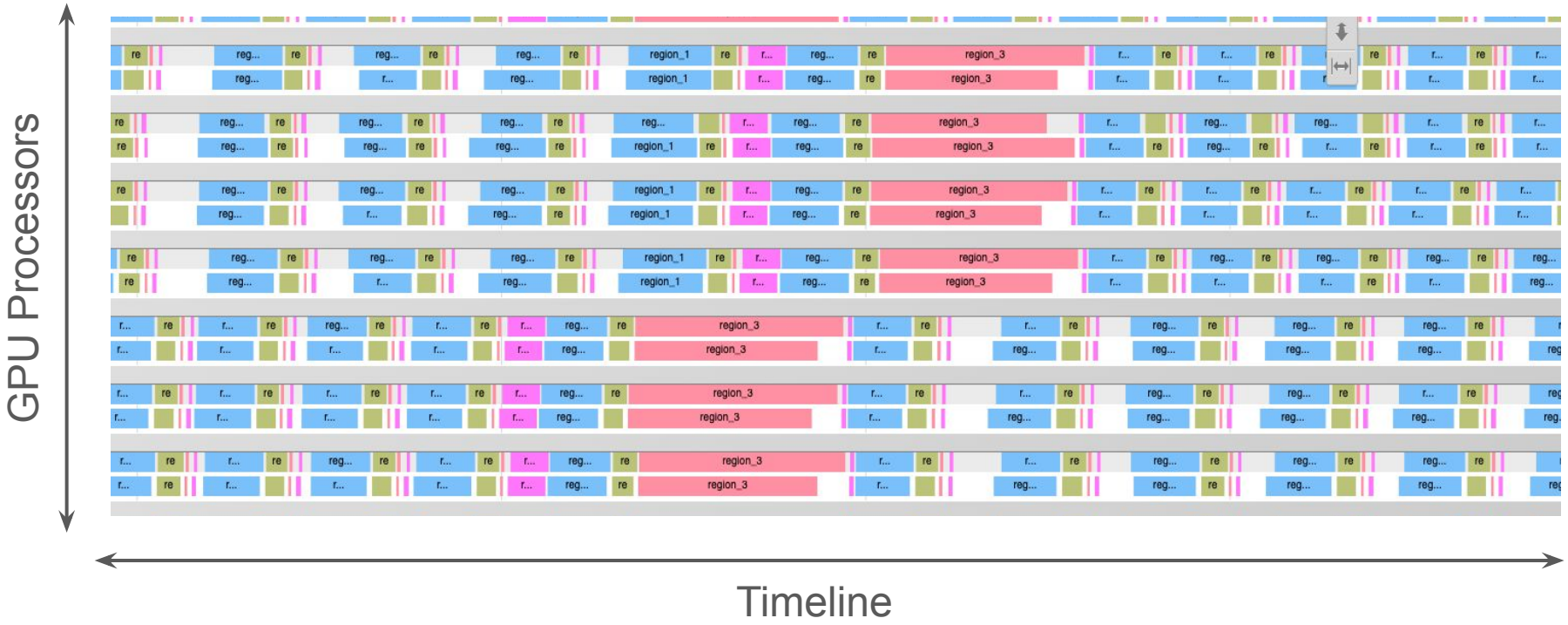
Use Cases

- Develop a custom “mode”
  - Fine-grained latency measurement for Triton IRs
    - Software pipelining
    - Warp specialization
- Associate profile data with compiler to build your own tools
  - Profiler-guided optimization
  - Collect and visualize values distribution of tensors

Triton FA-GEMM Memory-Compute Overlapping

- Nvidia CUTLASS library for high performance GEMMs uses warp specialization and a producer/consumer model to overlap memory and tensor core ops
  - Named barriers and warp specialization type features
- AMD Composable Kernels (CK) high performance GEMM library uses explicit setting of wave priority and compiler scheduling intrinsics
- Incorporating and optimizing of these methods into Triton requires advanced intra-kernel timing correlated with warp/wave, SM/CU IDs
  - Compiler based framework allows us to insert timestamps correlated to upper level data flow operations (e.g. loops and accumulate ops) and profile within the MLIR level passes and framework.
  - Using intra-kernel tracing we can identify areas to reorder instructions within the loop to get optimal overlapping of data movement (e.g. loads and stores) and compute (e.g. matrix instructions).

Fine-grained GPU Trace



Questions?