

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

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

```
186 @triton.jit
187 def matmul_kernel(
188     # Pointers to matrices:
189     a_ptr, b_ptr, c_ptr,
190     # Matrix dimensions
191     M, N, K,
192     # The stride variables represent how much to increase the ptr by when moving by 1
193     # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
194     # by to get the element one row down (A has M rows).
195     stride_am, stride_ak,
196     stride_bk, stride_bn,
197     stride_cm, stride_cn,
198     # Meta-parameters
199     BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
200     GROUP_SIZE_M: tl.constexpr,
201     ACTIVATION: tl.constexpr,
202 ):
203     """Kernel for computing the matmul C = A x B.
204     A has shape (M, K), B has shape (K, N) and C has shape (M, N)
205     """
206     # -----
207     # Map program ids `pid` to the block of C it should compute.
208     # This is done in a grouped ordering to promote L2 data reuse.
209     # See above `L2 Cache Optimizations` section for details.
210     pid = tl.program_id(axis=0)
211     num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
212     num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
213     num_pid_in_group = GROUP_SIZE_M * num_pid_n
214     group_id = pid // num_pid_in_group
215     first_pid_m = group_id * GROUP_SIZE_M
216     group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
217     pid_m = first_pid_m + (pid % group_size_m)
218     pid_n = (pid % num_pid_in_group) // group_size_m
```

MNK GEMM

Many layers exist between the Python code and the eventually generated ISA

```
graph TD
    subgraph MLIR_Lowering_Pipeline [MLIR Lowering Pipeline]
        A[Triton kernel DSL (Python-like)] --> B[Triton IR]
        B --> C[Triton GPU IR]
        C --> D[LLVM IR]
    end
    D --> E[AMDGCN ISA (LLVM AMDGPU Backend)]
```

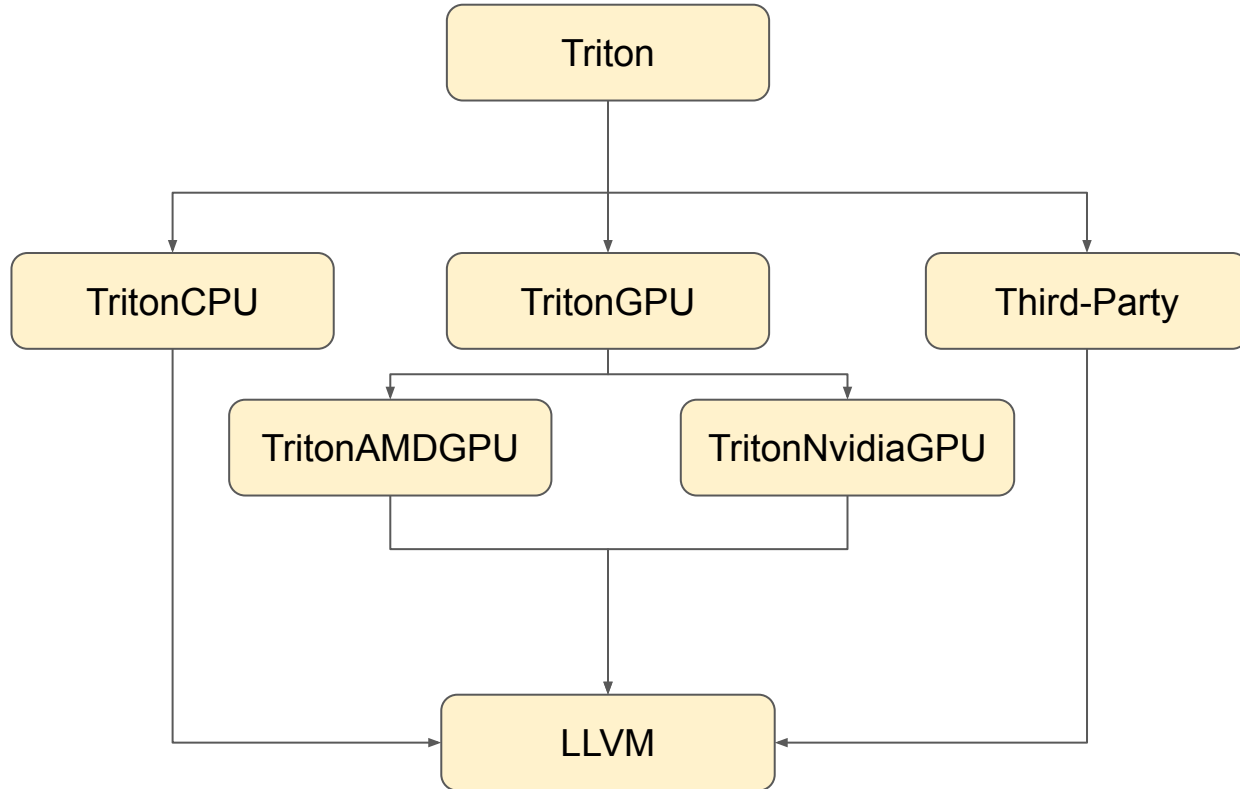
# Optimizing Code on Modern GPUs

- 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

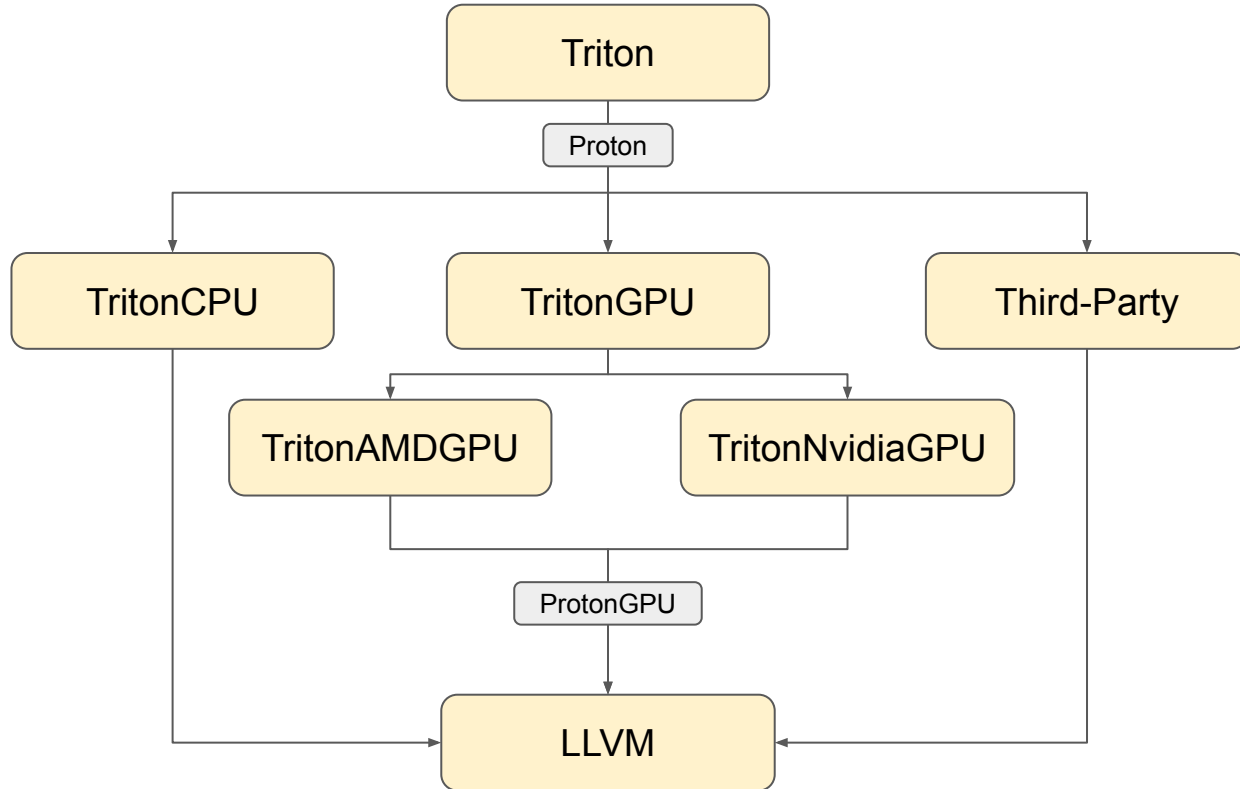
# Custom Instrumentation: Beyond CUPTI & RocTracer

- 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

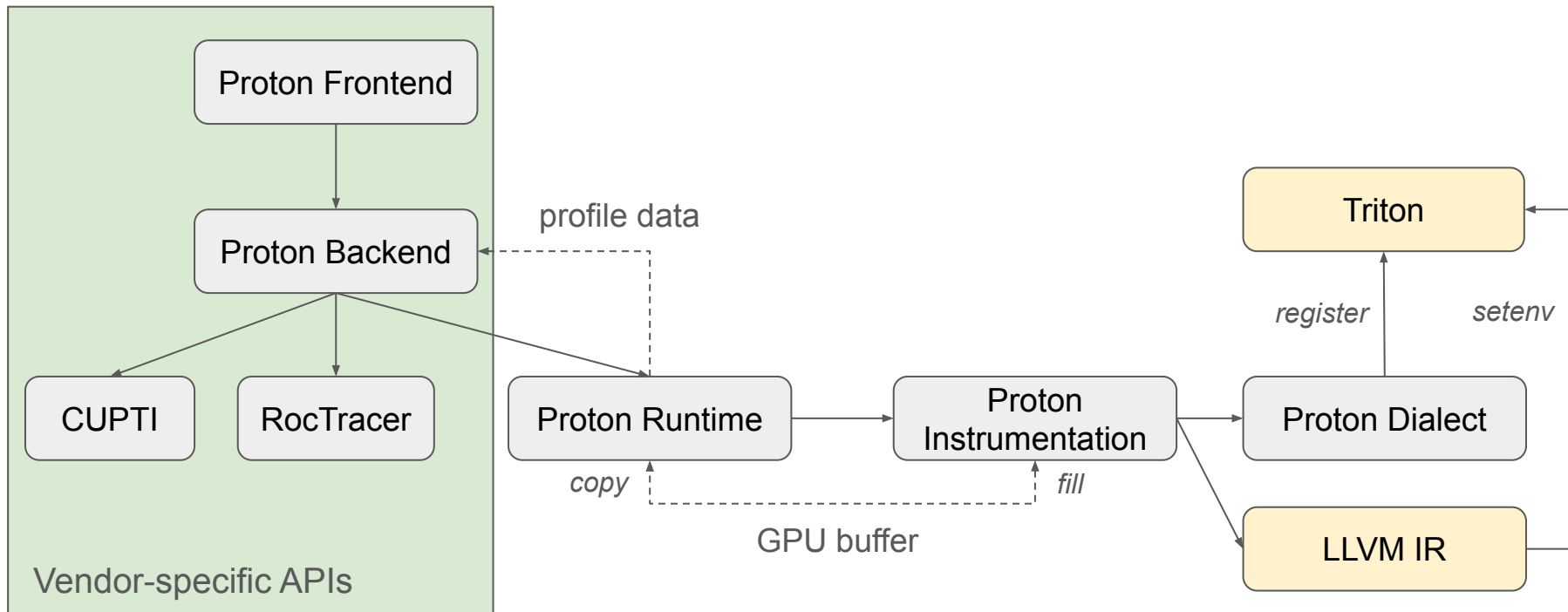
# Dialect Overview



# 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

```
1 import torch
2 import pytest
3 import pathlib
4
5 import triton
6 import triton.language as tl
7
8 @triton.jit
9 def add_kernel(
10     x_ptr,
11     y_ptr,
12     output_ptr,
13     n_elements,
14     BLOCK_SIZE: tl.constexpr,
15 ):
16     pid = tl.program_id(axis=0)
17     block_start = pid * BLOCK_SIZE
18     offsets = block_start + tl.arange(0, BLOCK_SIZE)
19     mask = offsets < n_elements
20     x = tl.load(x_ptr + offsets, mask=mask)
21     y = tl.load(y_ptr + offsets, mask=mask)
22     output = x + y
23     tl.store(output_ptr + offsets, output, mask=mask)
24 torch.manual_seed(0)
25 size = 2**12
26 x = torch.rand(size, device='cuda')
27 y = torch.rand(size, device='cuda')
28 output = torch.empty_like(x)
29 n_elements = output.numel()
30 grid = (1, 1, 1)
31 add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
```

```
1 import torch
2 import pytest
3 import pathlib
4
5 import triton
6 import triton.language as tl
7 import triton.profiler.language as pl
8
9 @triton.jit
10 def add_kernel(
11     x_ptr,
12     y_ptr,
13     output_ptr,
14     n_elements,
15     BLOCK_SIZE: tl.constexpr,
16 ):
17     pid = tl.program_id(axis=0)
18     block_start = pid * BLOCK_SIZE
19     offsets = block_start + tl.arange(0, BLOCK_SIZE)
20     mask = offsets < n_elements
21     x = tl.load(x_ptr + offsets, mask=mask)
22     pl.record(True, 0)
23     y = tl.load(y_ptr + offsets, mask=mask)
24     pl.record(False, 0)
25     output = x + y
26     tl.store(output_ptr + offsets, output, mask=mask)
27 torch.manual_seed(0)
28 size = 2**12
29 x = torch.rand(size, device='cuda')
30 y = torch.rand(size, device='cuda')
31 output = torch.empty_like(x)
32 n_elements = output.numel()
33 grid = (1, 1, 1)
34 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"
```

Start recording

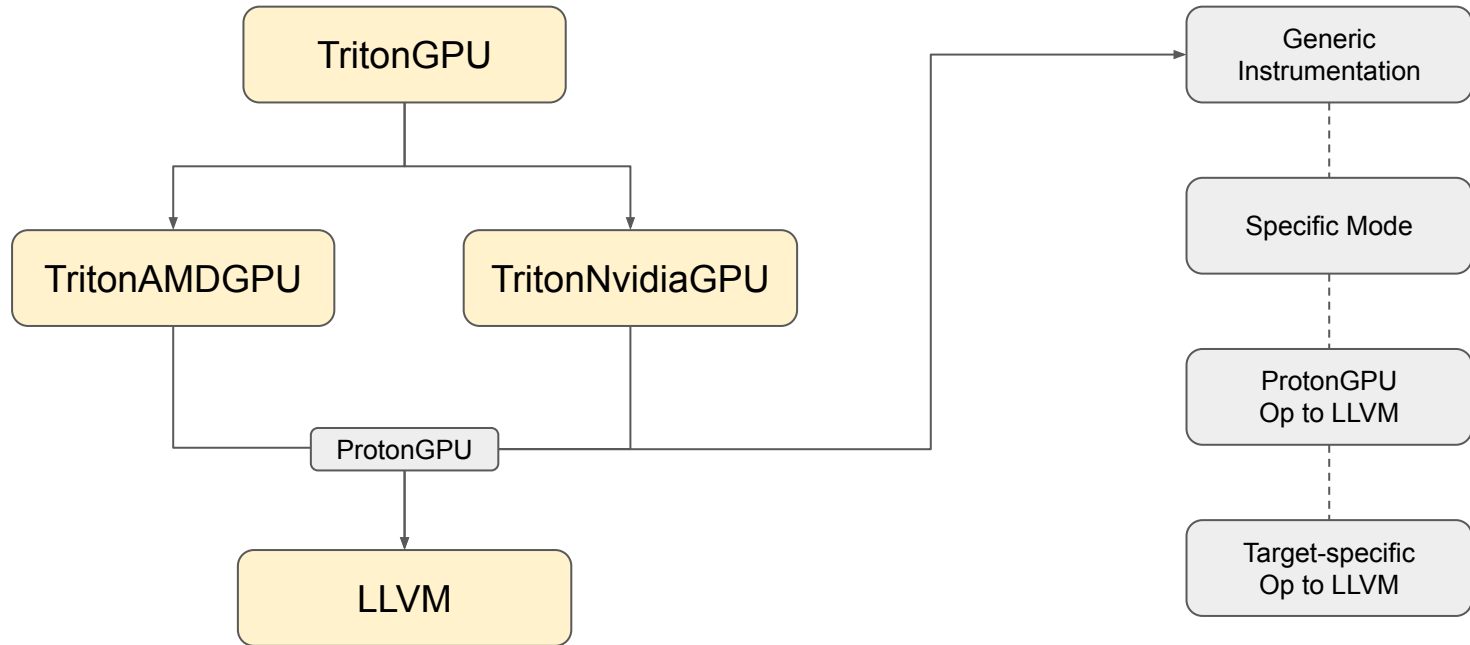
```
1 proton.record start "reduce"  
2 %b = "tt.reduce" (%v) ({  
3   ^bb0(%arg0: f32, %arg1: f32):  
4     %add = arith.addf %arg0, %arg1 : f32  
5     tt.reduce.return %add : f32  
6   }) {axis = 1 : i32} : (tensor<1x2x4xf32>) → tensor<1x4xf32>  
7 proton.record end "reduce"
```

Stop recording

# 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

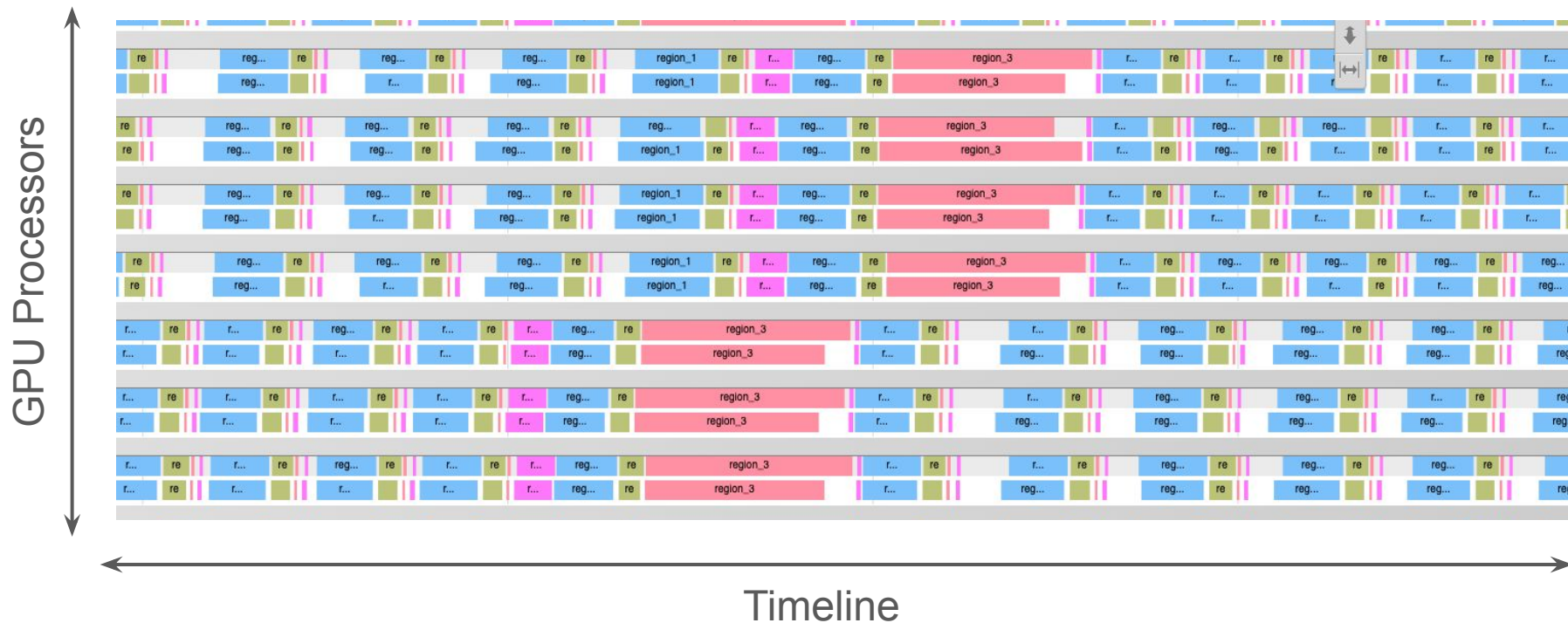
- Develop a custom “mode”
  - Fine-grained latency measurement for Triton IRs
    - Software pipelining
    - Warp specialization
- Associate profile data with compiler to build your own tools
  - Profiler-guided optimization
  - Collect and visualize values distribution of tensors

# 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?