Proton: Adaptive and Lightweight Profiling for Deep Learning Workloads

Corbin Robeck, Yuanwei Fang, **Keren Zhou**
kzhou6@gmu.edu

AI Applications

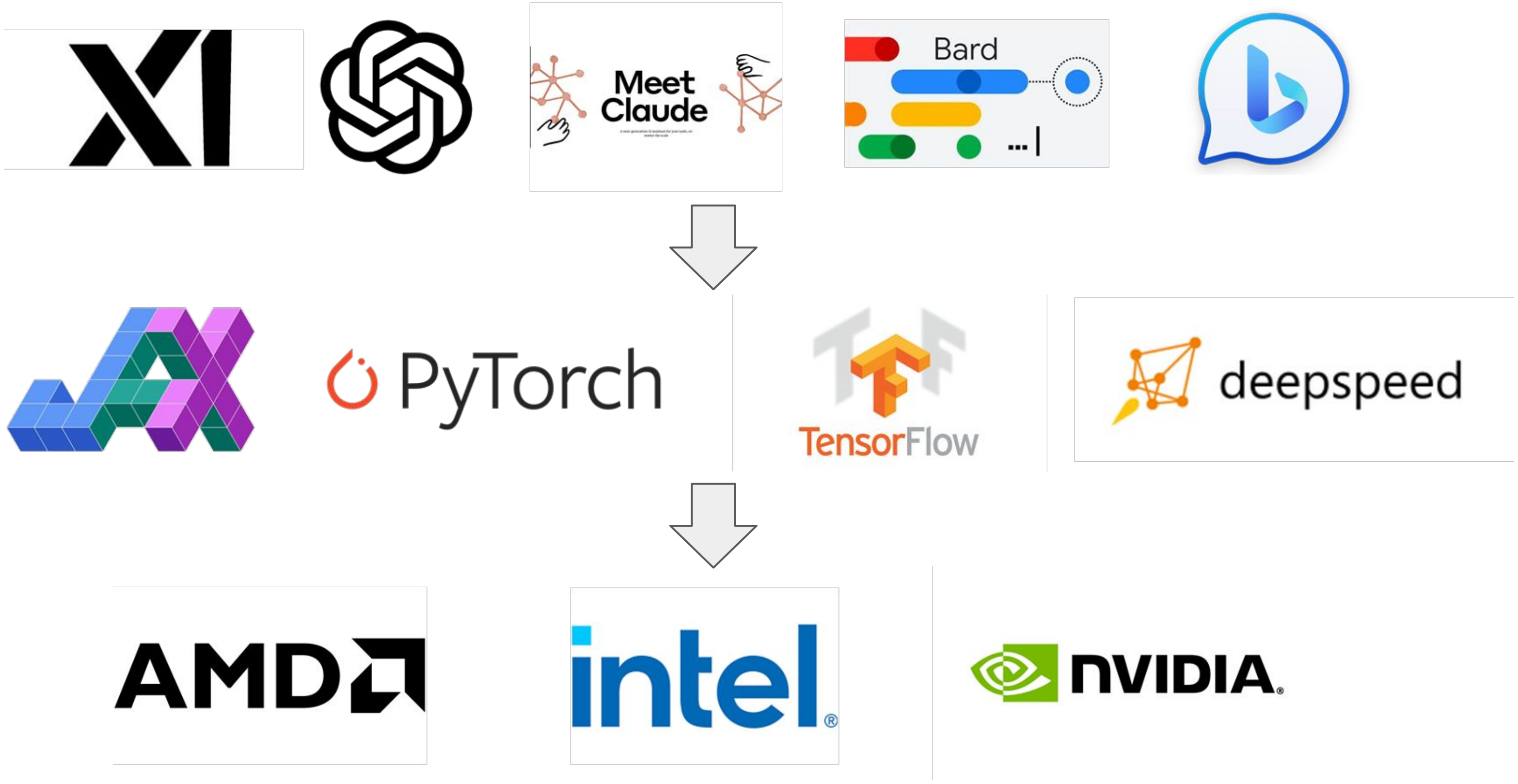
Grok

Claude

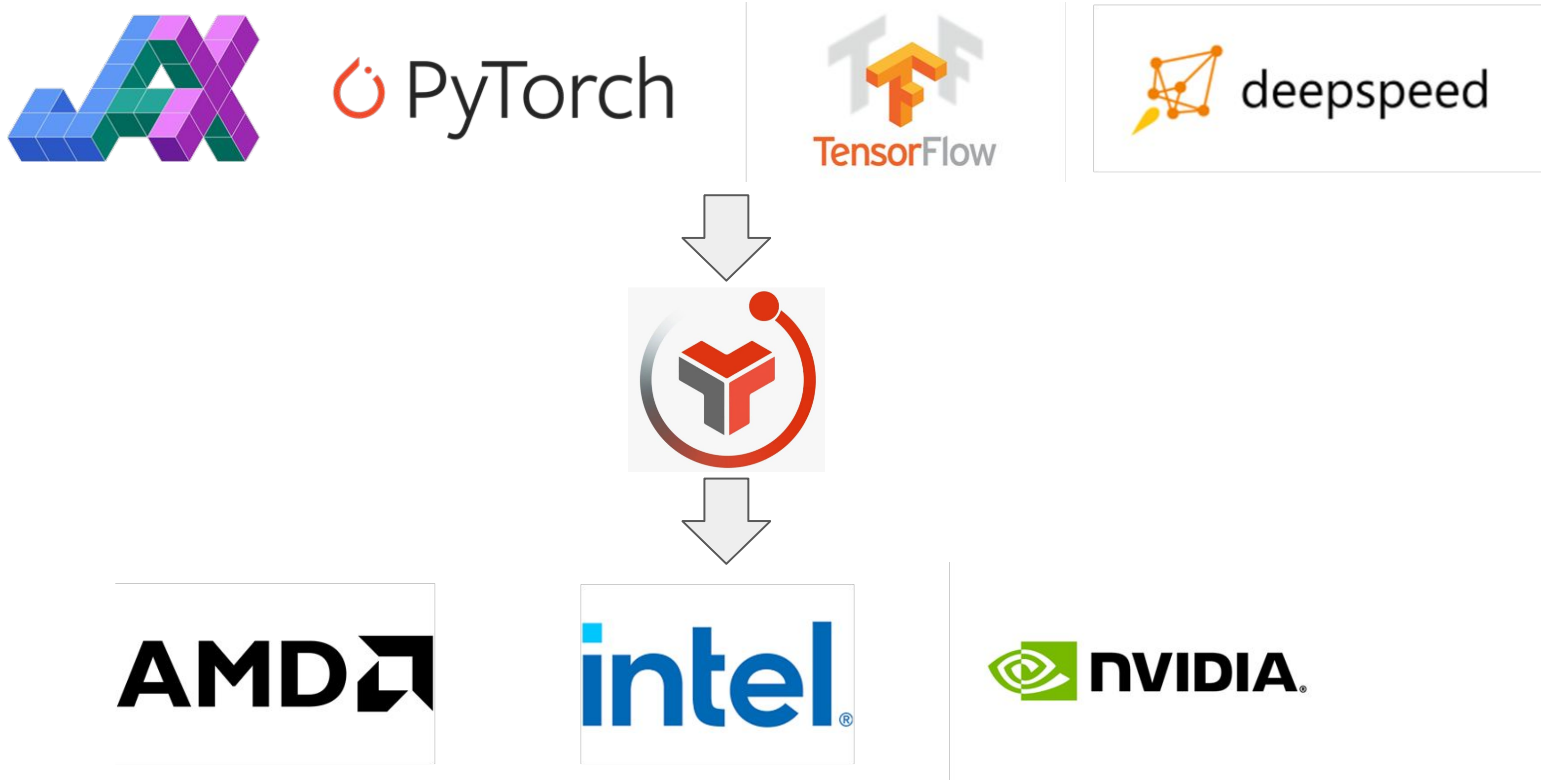
Midjourney

runway

AI System Software Stack

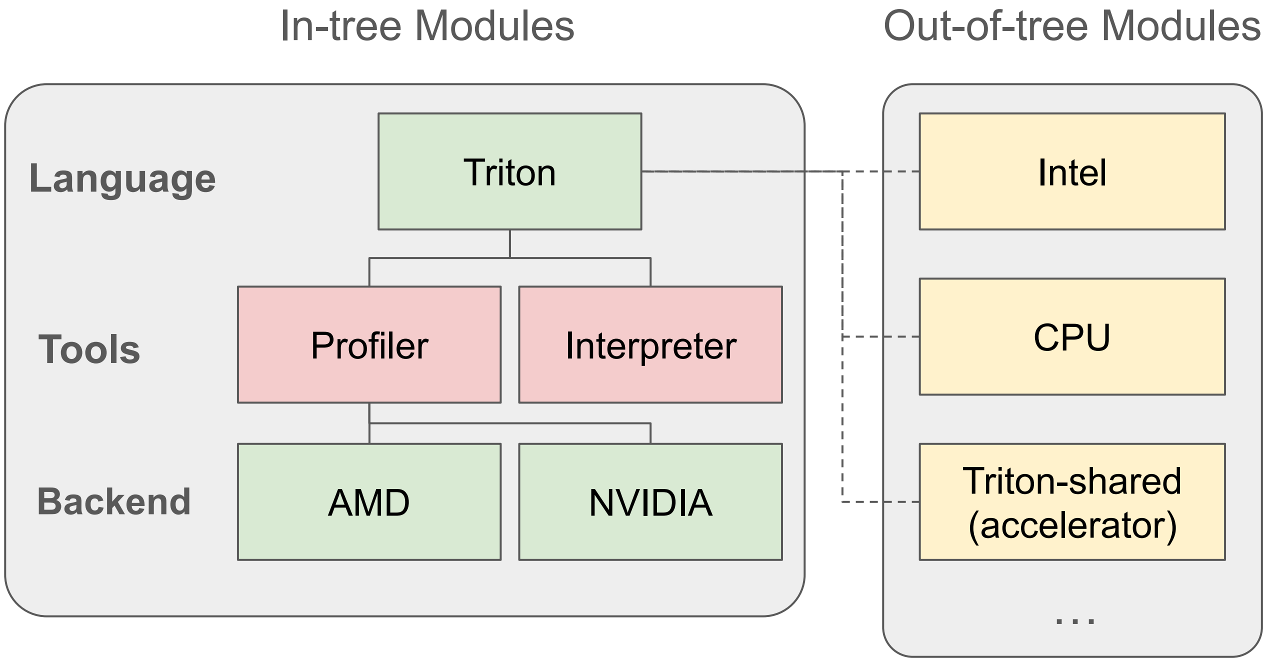


Why Triton?



Triton

Triton Modules



Triton Language

- Python-like language designed for high flexibility and performance in deep learning applications
  - Support tensor interface similar to PyTorch
  - Uses Python-like syntax
- Compared to CUDA/ROCm, Triton simplifies GPU programming
  - Only requiring knowledge that a kernel is divided into multiple blocks (Triton programs)
  - Most underlying details are handled by the compiler

A Simple Triton Program

```
z: dim0 x dim1 = x: dim0 x dim1 + y: dim0 x dim1
```

Kernel decorator

Programming model

Creation ops

Memory ops

vecAdd

```python
@triton.jit
def add_kernel(x_ptr, y_ptr, z_ptr, dim0, dim1,
               BLOCK_DIM0: tl.constexpr, BLOCK_DIM1: tl.constexpr):
    pid_x = tl.program_id(axis=0)
    pid_y = tl.program_id(axis=1)
    block_start = pid_x * BLOCK_DIM0 * dim1 + pid_y * BLOCK_DIM1
    offsets_dim0 = tl.arange(0, BLOCK_DIM0)[:,None]
    offsets_dim1 = tl.arange(0, BLOCK_DIM1)[None, :]
    offsets = block_start + offsets_dim0 * dim1 + offsets_dim1
    masks = (offsets_dim0 < dim0) & (offsets_dim1 < dim1)
    x = tl.load(x_ptr + offsets, mask=masks)
    y = tl.load(y_ptr + offsets, mask=masks)
    output = x + y
    tl.store(z_ptr + offsets, output, mask=masks)
```

Proton for Kernel Programmers

Proton (A **Pro**filer for Tri**ton**)

- Provide a quick, intuitive, and simple way to check kernel performance
  - Open source
  - Multiple vendor GPUs
  - Flexible metrics collection
  - Hardware metrics
  - Software metrics
- Call path profiling
- Timeline tracing*

Proton *vs* Nsight Systems *vs* Nsight Compute

Tool	Nsys	NCU	Proton
Overhead	Up to 3x	Up to 1000x	Up to 1.5x
Profile size	Large	Large	Tiny (<1MB)
Profiling targets	NVIDIA GPUs, CPUs	NVIDIA GPUs	NVIDIA and AMD GPUs
Granularity	Kernels	Kernels and instructions	Regions, kernels and instructions
Metrics	GPU time GPU utilization CPU samples	A complete set of metrics from hardware counters	GPU time GPU instruction samples **User-defined metrics**
Triton hooks	N/A	N/A	Support

User Interface

- Lightweight source code instrumentation
  - Profile start/stop/finalize
  - Scopes
  - Hooks
- Command line
  - `python -m proton main.py`
  - `proton main.py`

Start/Stop/Finalize Profiling

- Profile only interesting regions
    - `proton.start(profile_name: str) -> session_id: int`
    - `proton.finalize()`
- Skip some regions, but accumulate to the same profile
    - `session_id = proton.start(...)`
    - `proton.deactive(session_id)`
    - `… # region skipped`
    - `proton.activate(session_id)`

Scopes

- A user-defined region with semantic information
  - Initialization
  - Forward
  - Backward
- with proton.scope(name)

Metrics

- Hardware metrics
  - Come from profiling substrates (e.g., CUPTI)
    - Kernel time
    - Instruction samples
- User-defined metrics
  - Come from users
    - Flops
    - Bytes
    - Tokens

Instruction Sampling

- For large functions, we need fine-grained insights about which lines/IRs/instructions are expensive
- Instruction sampling is an experimental feature we're developing to support this goal
    - It's called *pc sampling* using NVIDIA's terminology

Case Study: Persistent Matmul Optimization

- We use scopes to annotate
  - Matmul shapes: matmul [M_N_K]
  - Autotuned configurations: <autotune>
  - cuBLAS/Torch/Triton kernels
- We use hooks to annotate
  - Grid dimensions
  - Number of warps
  - Number of stages

Case Study: Persistent Matmul Optimization

[Pipeliner] Enable automatic loop fusion by Mogball · Pull Request #5726 · triton-lang/triton

```
root@dev-0:~/code/triton$ python python/tutorials/09-persistent-matmul.py
M=32, N=32, K=32 verification naive vs: torch: ✅ cublas: ✅ persistent: ✅ TMA persistent: ✅ Tensor descriptor persis
M=8192, N=8192, K=512 verification naive vs: torch: ✅ cublas: ✅ persistent: ✅ TMA persistent: ✅ Tensor descriptor p
273.146 4025.362 ROOT
├─ nan 0.031 _ZN2at6native18elementwise_kernelILi128ELi4EZNS0_22gpu_kernel_impl_nocastIZZZNS0_23direct_copy_kernel_cudaER
├─ nan 0.027 _ZN2at6native54_GLOBAL__N__a236ace4_21_DistributionNormal_cu_0c5b6e8543distribution_elementwise_grid_stride_
├─ 283.506 2666.310 cublas [M=8192, N=8192, K=512]
│  └─ nan 2666.310 sm90_xmma_gemm_f16f16_f16f32_f32_tn_n_tilesize128x128x64_warpgroupsize1x1x1_execute_segment_k_off_kern
├─ 223.326 307.709 matmul_kernel [M=8192, N=8192, K=512]
├─ 259.293 265.027 matmul_kernel_descriptor_persistent [M=8192, N=8192, K=512]
├─ 238.500 288.133 matmul_kernel_persistent [M=8192, N=8192, K=512]
├─ 258.738 265.594 matmul_kernel_tma_persistent [M=8192, N=8192, K=512]
└─ 295.529 232.531 torch [M=8192, N=8192, K=512]
   └─ nan 232.531 sm90_xmma_gemm_f16f16_f16f32_f32_tn_n_tilesize128x128x64_warpgroupsize1x1x1_execute_segment_k_off_kerne

Legend (Metric: tflop16/s (inc) Min: 223.33 Max: 295.53)
█ 288.31 - 295.53
█ 273.87 - 288.31
█ 259.43 - 273.87
█ 244.99 - 259.43
█ 230.55 - 244.99
█ 223.33 - 230.55
```

Flexible Performance Analysis

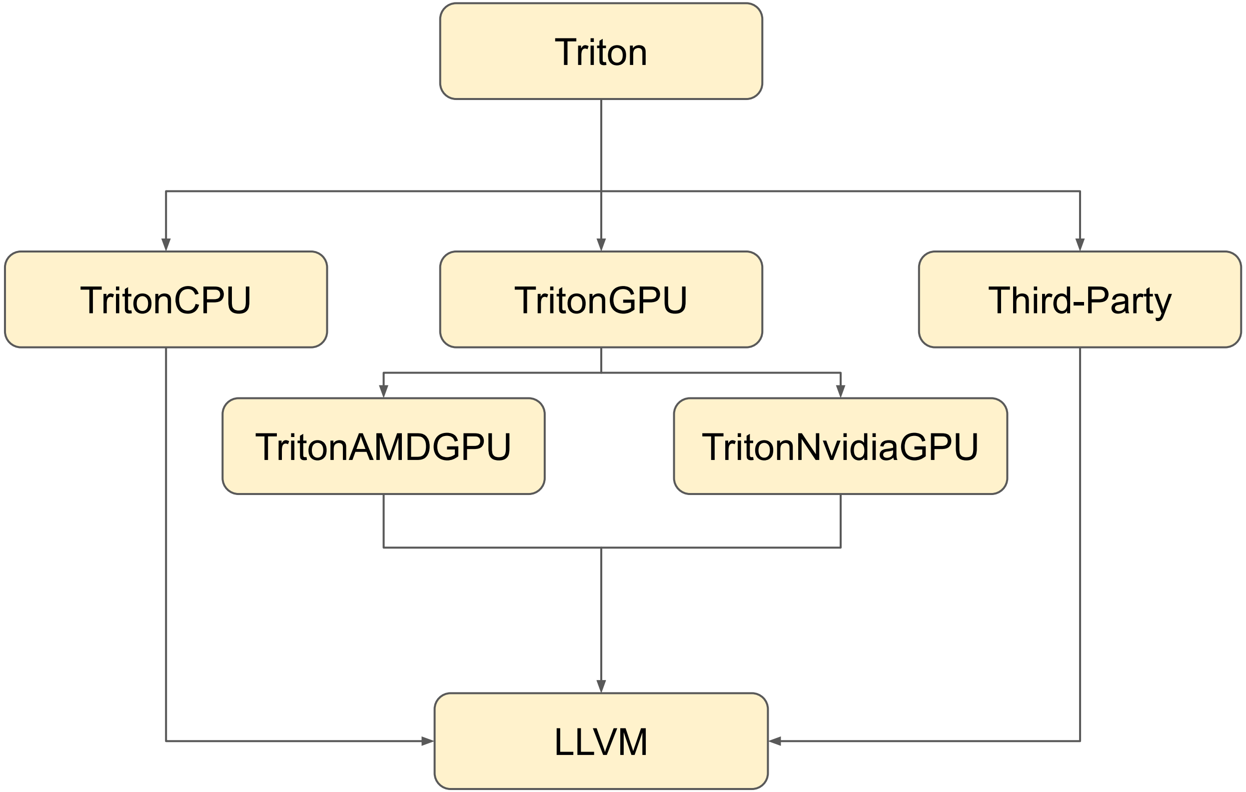
- Command line-based metrics derivation
  - `proton-viewer -m tflop/s tbyte/s`
  - `proton-viewer -diff profile0 profile1`
- Python-based profile analysis
  - Loads profiles as a Hatchet graph frame
    - Modify the graph
    - Extract hotspots
    - Merge multiple graphs
    - Derives insights at each node

Proton for Compiler Engineers

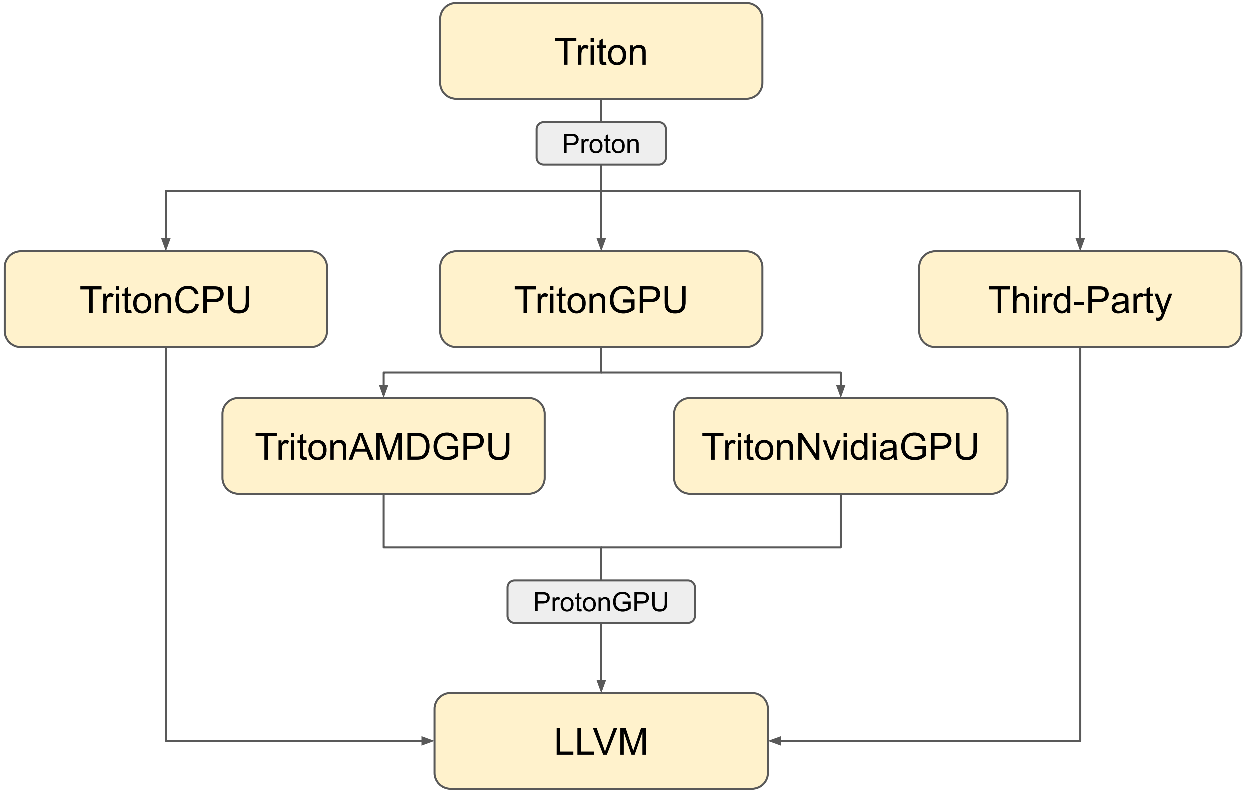
Custom Instrumentation: Beyond CUPTI & RocTracer

- 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/optimization across kernels
  - Extended metrics: Capture data unavailable through vendor tools

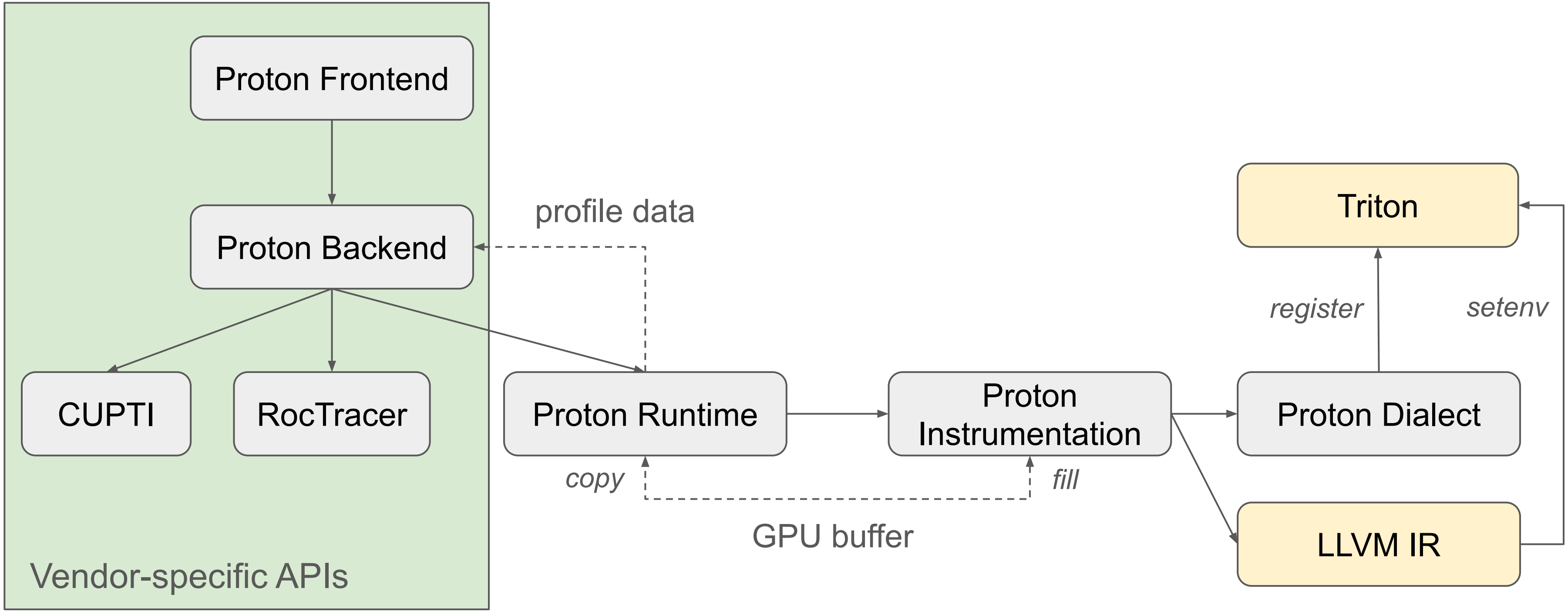
Dialect Overview



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

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 Instrumentation

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

reduce

Start recording

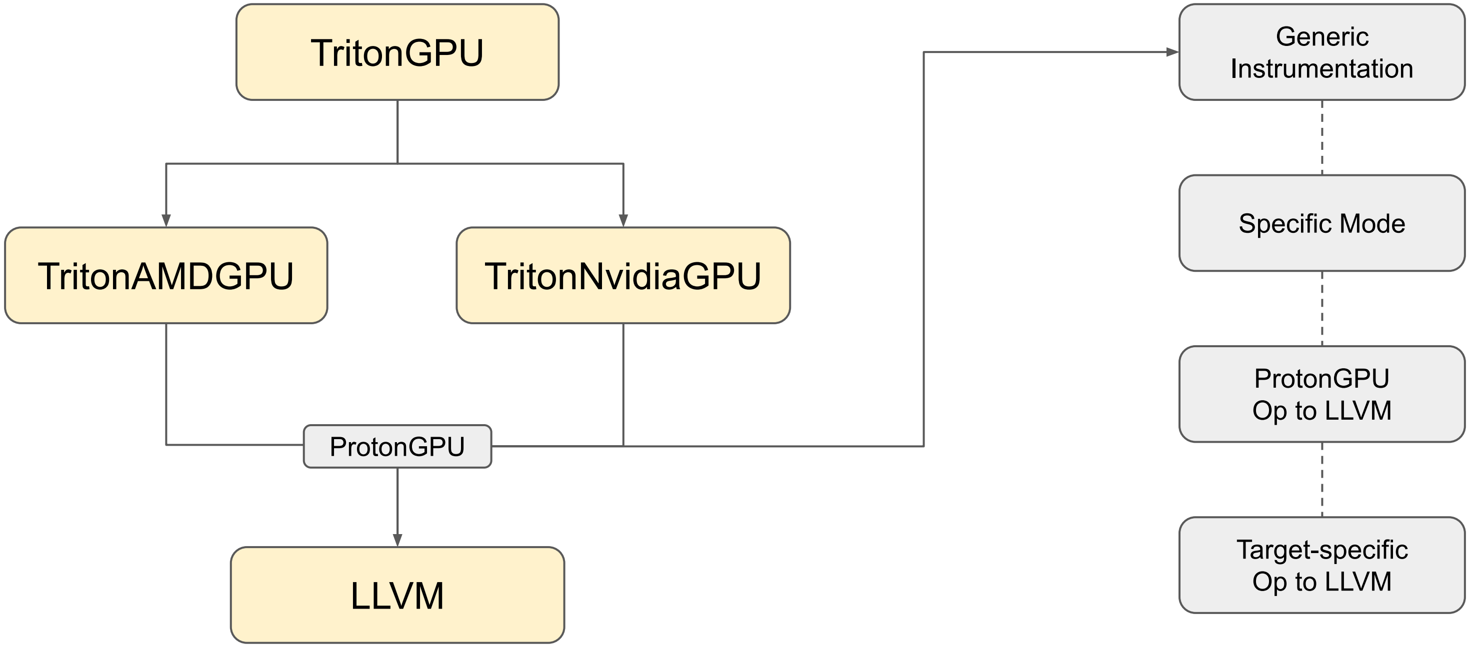
```
proton.record start "reduce"
%b = "tt.reduce" (%v) ({
  ^bb0(%arg0: f32, %arg1: f32):
    %add = arith.addf %arg0, %arg1 : f32
    tt.reduce.return %add : f32
}) {axis = 1 : i32}  : (tensor<1x2x4xf32>) → tensor<1x4xf32>
proton.record end "reduce"
```

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

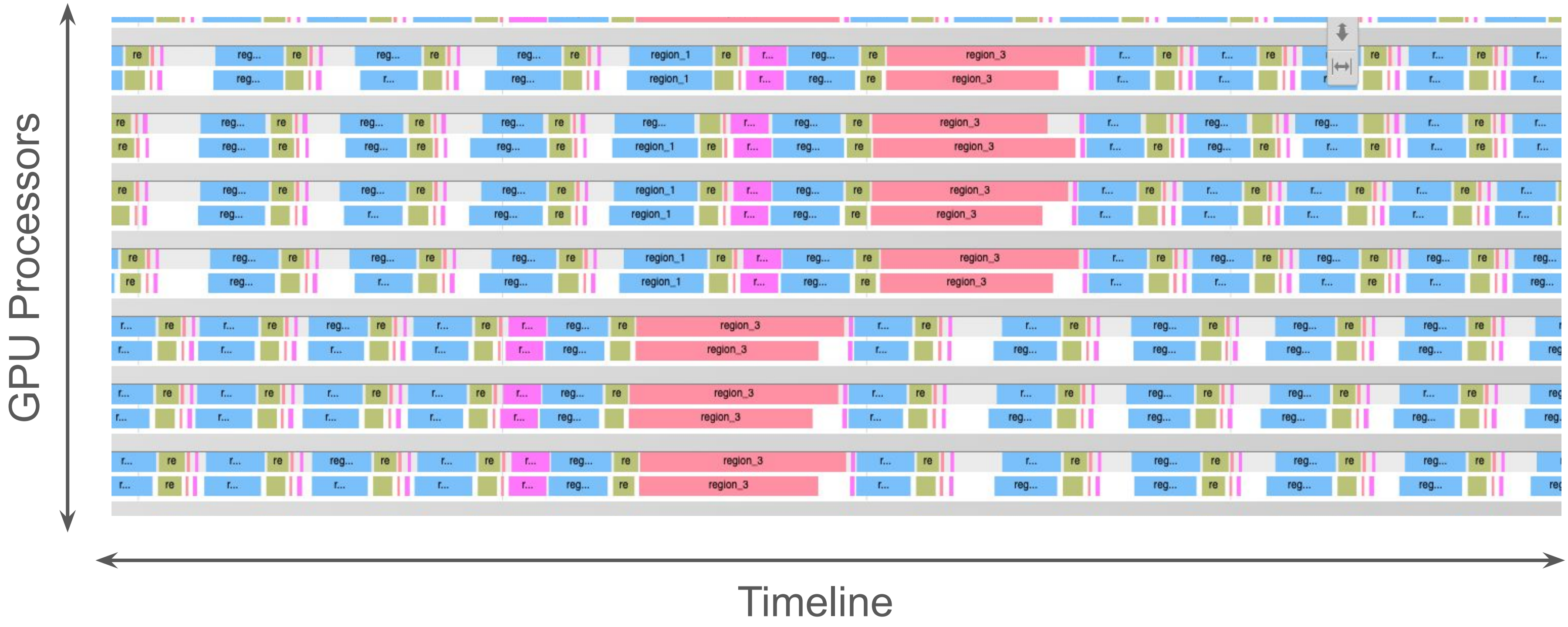
ProtonGPU to LLVM Lowering



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

Fine-grained GPU Trace



What's the Next

- **Release the warp specialization tracing mode**
- Support more backends and instrumentation modes
- Support inductor-compiled kernels
- We aim to avoid reinventing the wheel
    - Reimplementing functionalities that can be easily achieved using Nsight Compute or Nsight Systems

Triton Hook

- A way to compute and associate metrics with each Triton kernel launch
  - `@triton.jit(launch_metadata=metadata_fn)`
- `metadata_fn` is a callback function that
  - Takes three input arguments
    - Grid
    - Metadata
      - warps, stages, shared
    - Args
  - Returns a dictionary containing
    - Renamed kernel name
    - Other metric names and values

Instruction Sampling

- Sample an instruction on each active GPU SM every *N* cycles
- Each instruction is associated with a *stall* reason if available
    - Why the instruction was not issued
- “Low overhead” with regard to each kernel’s GPU time
- Available on NVIDIA, AMD and Intel GPUs

Viewer

- `proton-viewer` a call path visualization tool
- Load json data into `pandas`
- Render it on terminal using `hatchet`
  - LLNL-Hatchet: A flexible package for performance data analysis
  - Hatchet can also convert the format into other formats such as flamegraph
- `proton-viewer -h` for more information