Proton: Introduction and Development

Yuanwei Fang, Corbin Robeck, and Keren Zhou

Introduction

Background

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

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	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

Key APIs

- Profile only interesting regions
  - proton.start(profile_name: str) -> session_id: int
  - proton.finalize()
- Annotate a user defined region with semantic information
  - proton.scope(name: str, metrics: {})
- Rename Triton Kernels
  - @triton.jit(launch_metadata=metadata_fn)

Example

- We use scopes to annotate
  - Matmul shapes: matmul_M_N_K
  - Autotuned configurations: <autotune>
- Commands to run
  - proton/tutorials/matmul.py
  - proton-viewer -m time/ms -i ".*triton.*" ./matmul.hatchet

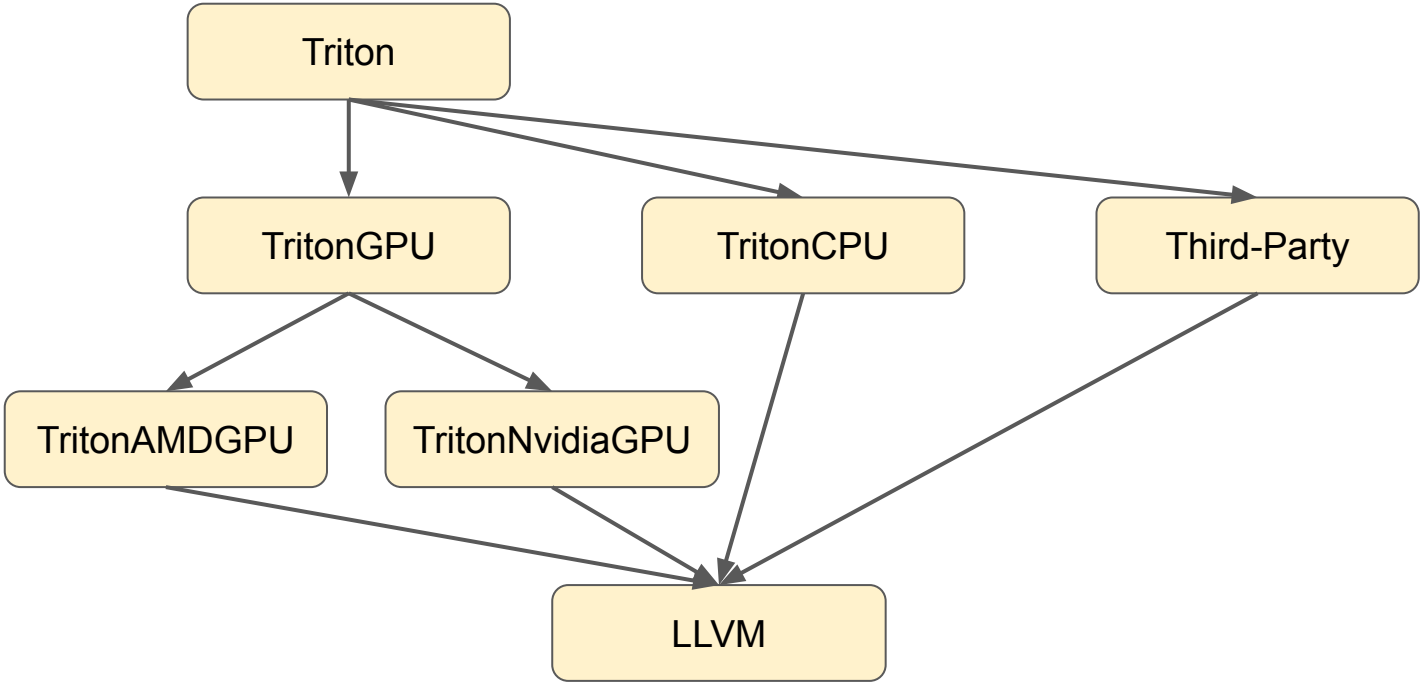
```
├─ 1090.280 matmul_1024_1024_1024
│  └─ 981.306 triton
│     ├─ 90.695 <autotune>
│     │  ├─ 18.325 matmul_<grid:128x1x1>_<cluster:1x1x1>_<warps:4>_<shared:36864>_<stages:4>
│     │  ├─ 14.102 matmul_<grid:256x1x1>_<cluster:1x1x1>_<warps:4>_<shared:30720>_<stages:4>
│     │  ├─ 8.593 matmul_<grid:32x1x1>_<cluster:1x1x1>_<warps:8>_<shared:98304>_<stages:3>
│     │  ├─ 32.475 matmul_<grid:512x1x1>_<cluster:1x1x1>_<warps:2>_<shared:24576>_<stages:5>
│     │  ├─ 8.307 matmul_<grid:64x1x1>_<cluster:1x1x1>_<warps:4>_<shared:49152>_<stages:4>
│     │  └─ 8.893 matmul_<grid:64x1x1>_<cluster:1x1x1>_<warps:4>_<shared:61440>_<stages:4>
│     ├─ 882.505 _ZN2at6native29vectorized_elementwise_kernelILi4ENS0_11FillFunctorIiEENS_6de
│     └─ 8.106 matmul_<grid:64x1x1>_<cluster:1x1x1>_<warps:4>_<shared:49152>_<stages:4>
```

On-going Development

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 Client Interface: Simplify development across different platforms
  - Extended Metrics: Capture data unavailable through vendor tools
- Use cases
  - Memory heat map generation to visualize performance bottlenecks
  - Tailored instrumentation for asynchronous matrix multiplication instructions

Dialect Overview



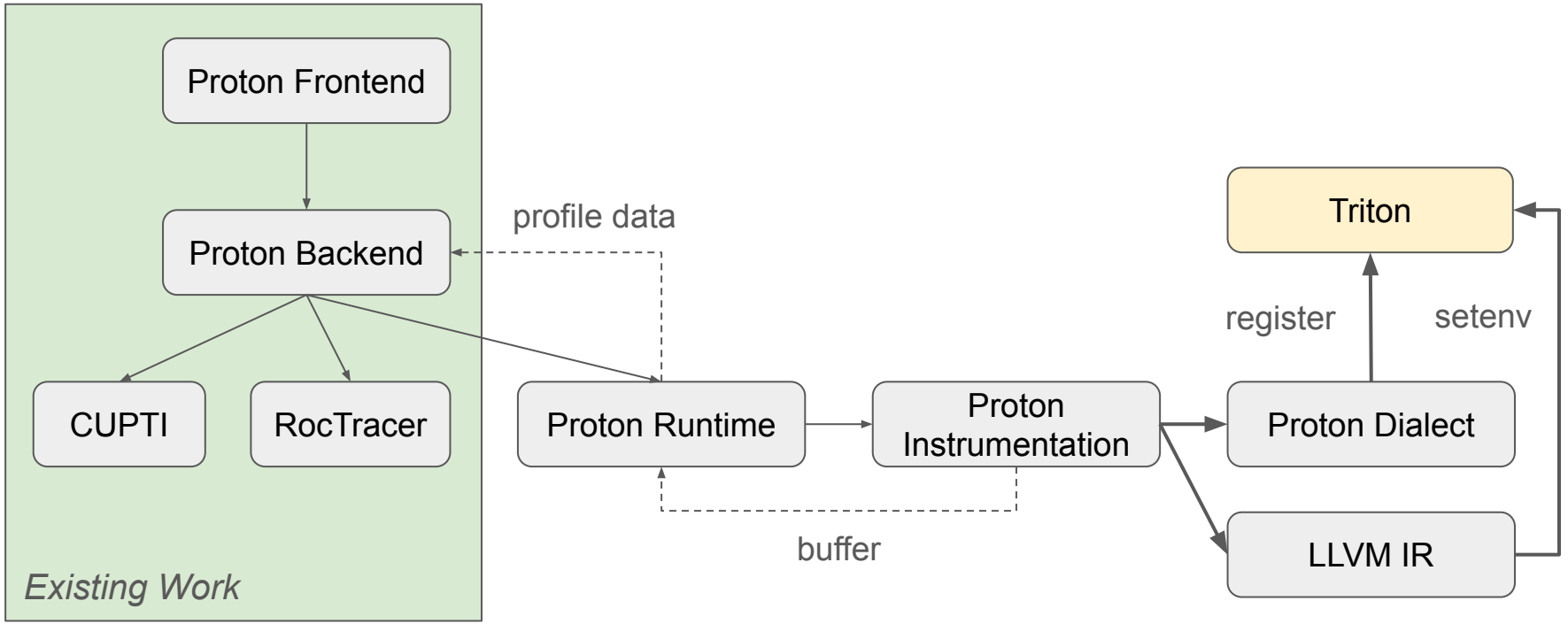
MLIR Instrumentation Infrastructure

- The standard MLIR instrumentation has been utilized already in Triton
  - PassInstrumentationCallbacks *instrCbPtr = nullptr;
  - PassInstrumentationCallbacks passInstrCb;
  - StandardInstrumentations standardInstr(mod->getContext(), /*DebugLogging*/ true);
- https://mlir.llvm.org/docs/PassManagement/#pass-instrumentation
  - runAfterPass
  - runBeforePass

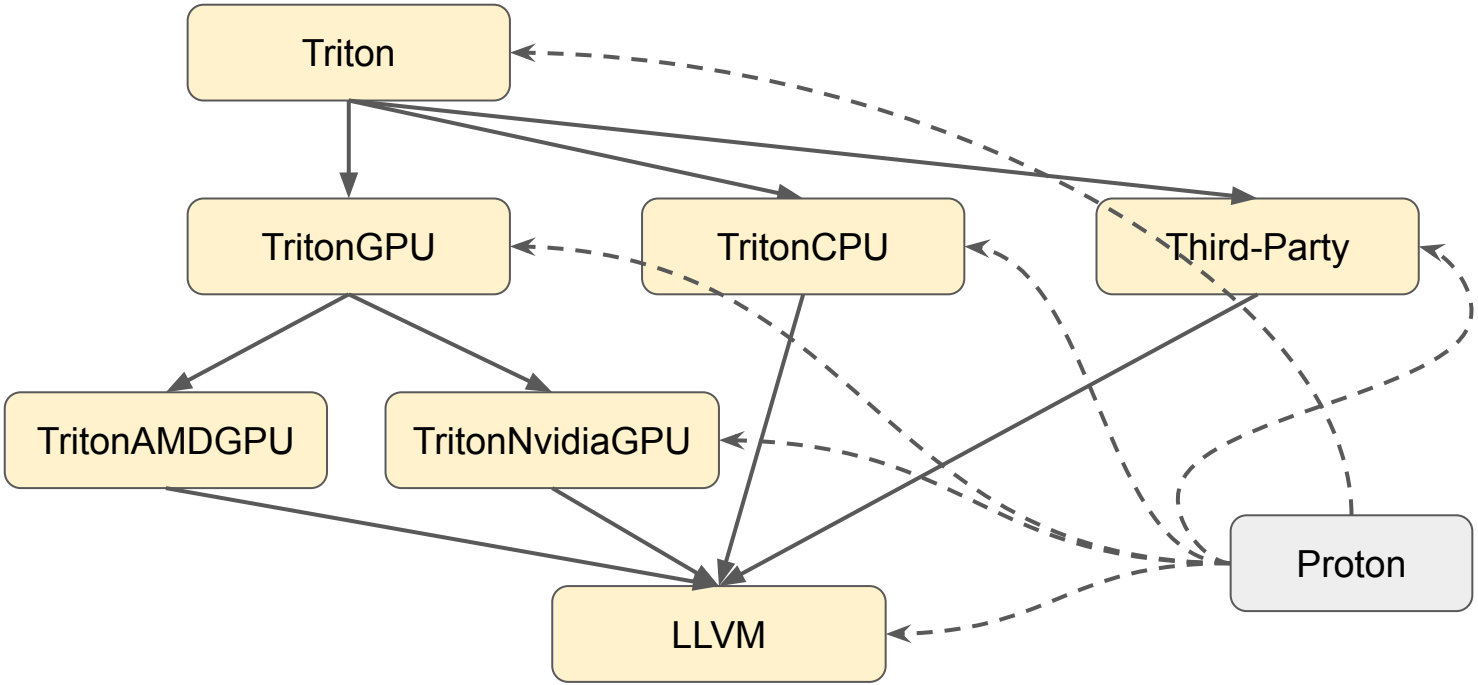
LLVM IR Instrumentation

- Dialect-free
- Allows users to write C/C++ callbacks
  - Valuable for fast debugging without going through the profiler
  - Allow third-party tools to easily inject Triton binaries
- Already supported in Triton with an environment variable
- **Defer introduction to Corbin**

Proposed Solution



Proton Dialect



Proton Dialect Ops

- Consist of common *MLIR* ops that can be utilized for different instrumentation clients
  - `Proton.InitOp`
    - Initialize a local buffer
  - `Proton.RecordOp`
    - Collect a record
  - `Proton.NotifyOp`
    - Notify the host that the buffer is full
  - `Proton.FinalizeOp`
    - Finalize the profiling
- **Defer introduction to Yuanwei**

Proton Dialect Scope and Usage

- Landscape
  - **API-based profiler**: Linux Perf API, Proton Dialect
  - **Instrumentation-based profiler**: Valgrind/Callgrind (runtime binary instrumentation), Proton instrumentation tool (compile-time instrumentation)
- Usage
  - **Triton IR level Latency Measurement**
    - Example:
      - Software pipelining
      - Warp specialization
      - GPU reverse engineering
  - **Build Your Own Tool**
    - Leverage compiler capability to extract your Triton program’s syntax
    - Customized proton-enabled MLIR passes with CLI
    - Example:
      - Multi-iteration async-wait visualizer
      - Profiler-guided optimization
      - Hybrid performance analyzer

Proton Instrumentation

- Parse instrumentation requests from users
- `proton.instrument(mode=mode_name)`
  - Patch all functions with the given mode
  - Maybe we can allow multiple modes to be used together
- `proton.instrument(fn, mode=xxx) / proton.restore(fn)`
  - From this point going on, patch/unpatch the specified triton function

Proton Runtime

- `proton.profile(..., backend="instrumentation", …)`
  - The cupti backend can be used together with the instrumentation backend
- `ProtonInstrumentationEnable`
  - Invoke the instrumentation engine
- `ProtonBufferInit`
  - Manage host/device GPU buffers
- `ProtonCallbackRegister`
  - When the device buffer is full, copy back the buffer to the host
- `ProtonActivityKind`
  - Define data structures for each type of instrumentation modes

Changes in Triton

- Register the Proton Dialect
- Add an instrumentation mode string in the hash key
- Link Proton with MLIR dialects

Put it together

- An end-to-end use case will be like the following
- proton.start(name=”profile1”, backend=”cupti”, …)
  - Profile time and hardware counter metrics
- proton.start(name=”profile2”, backend=”instrumentation”, …)
- proton.instrument(fn, mode=”mode1”)
  - Profile custom metrics from mode1
  - These metrics can be profiled simultaneously and aggregated to either the same or different profiles