Tools for Triton

Keren Zhou

Assumptions

- We assume
  - You have heard of Triton
  - You are familiar with Python and PyTorch

- We do not assume
  - You are a GPU expert
  - You are a compiler expert

Outcomes

- Understand the challenges of debugging and profiling Triton programs
- Learn how to use the Triton interpreter for debugging
- Gain some familiarity with using Proton for performance analysis

Outline

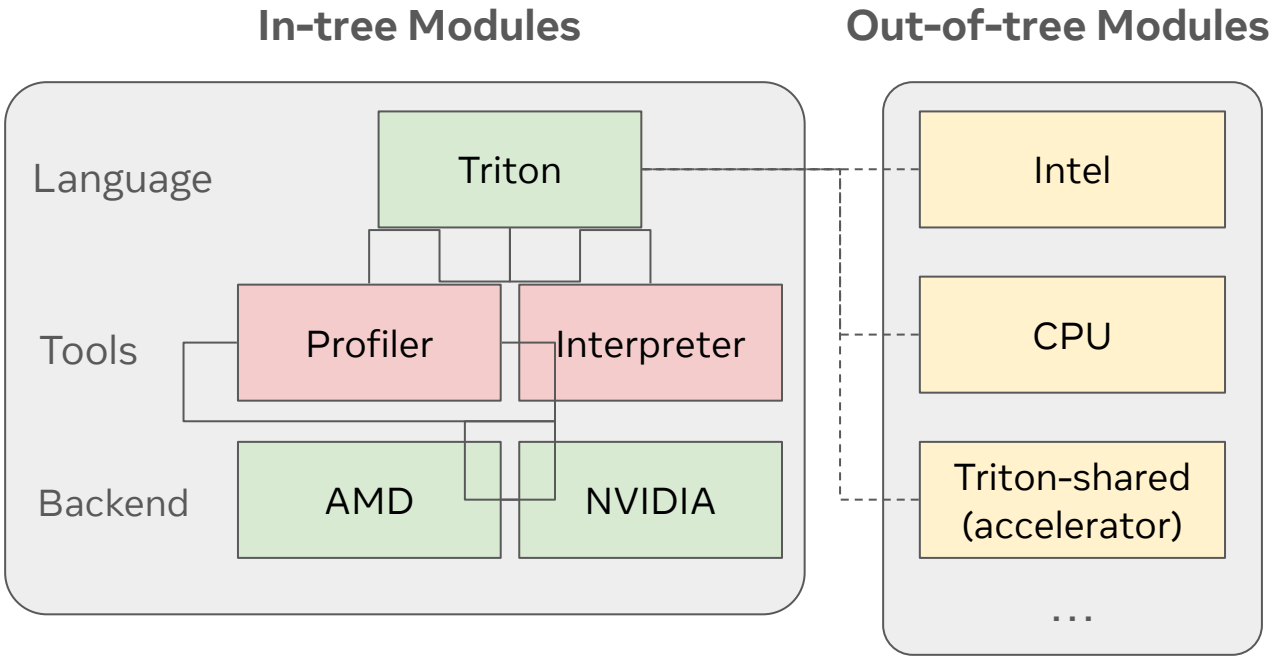
- Triton Review
- Interpreter
- Proton
- Proton: Instruction Sampling

Triton Review

What is Triton

- A Python-like language
- A JIT compiler
- A PyTorch backend
- A set of MLIR dialects
- An organization
- A community

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)
```

Outline

- Triton Review
- Interpreter
- Proton
- Proton: Instruction Sampling

Interpreter

Debugging GPU Execution is Difficult

vecAdd

```
1  __global__ void vecAdd(int *A, int *B, int *C) {
2    int i = blockDim.x * blockIdx.x + threadIdx.x;
3
4    // Error: trying to access the 385th elements
5    C[i] = A[i] + B[i];
6  }
7
8  int *d_A, *d_B, *d_C;
9  cudaMalloc((void **)&d_A, 384);
10 cudaMalloc((void **)&d_B, 384);
11 cudaMalloc((void **)&d_C, 384);
12 vecAdd<<<4, 128>>>(d_A, d_B, d_C);
```

Debugging GPU Execution is Difficult

RuntimeError: CUDA error: an illegal memory access was encountered

Greg Brockman 
@gdb

Went to sleep at 3:30a last night trying to track down a nasty CUDA illegal memory access bug, woke up early to keep digging, & just now pushed a fix. Hard to beat the endorphin rush of doing battle with a tricky bug and coming out on top.

5:06 pm · 26 Jun 2022

Common Errors in GPU Programming

- Out-of-bounds memory access
- Global/shared memory data race
  - Multiple threads access the same locations
- Improper synchronization
  - Deadlock/unexpected results
- Memory leaks
  - Tensors are not freed after use
- Misaligned memory access
  - Vectorized memory instructions report errors
- …

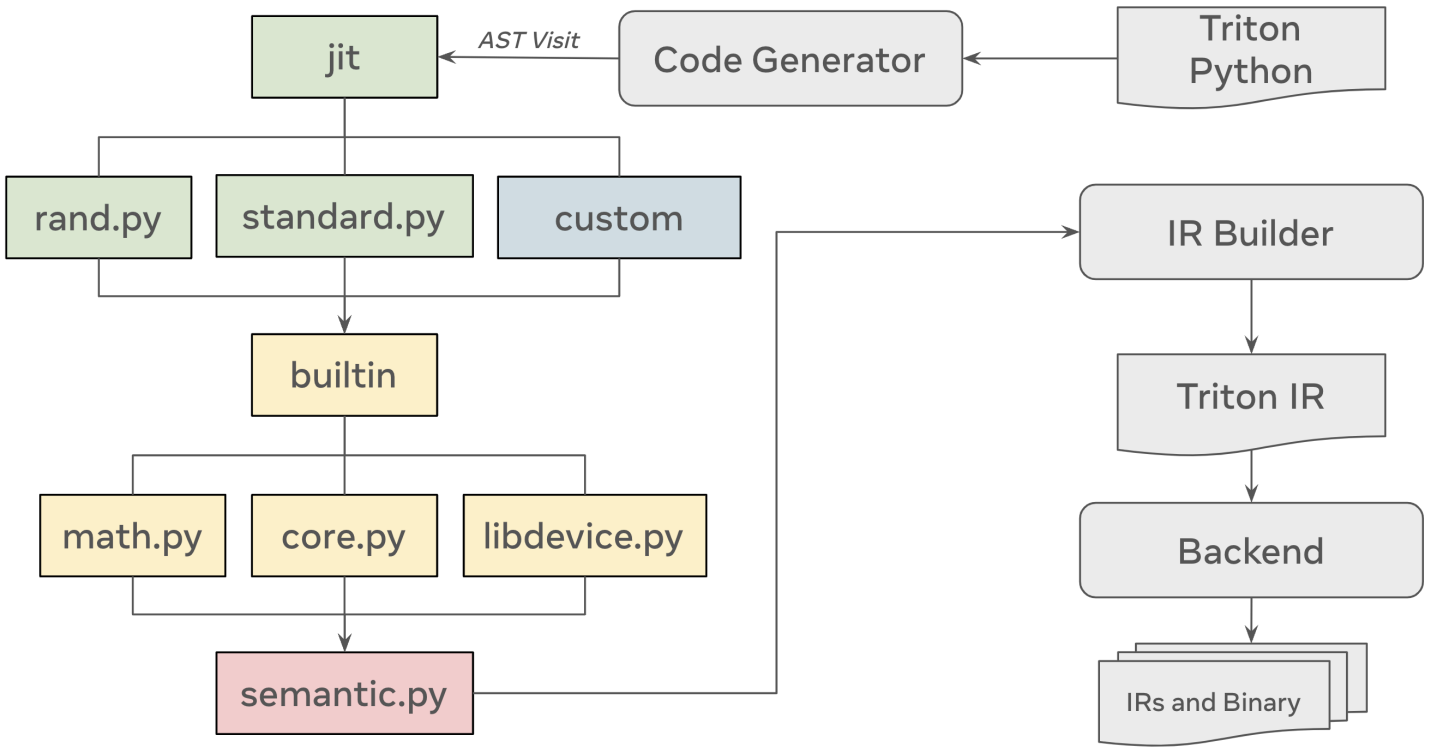
Debugging Triton Programs is Still Not Easy

- Launch parallel programs instances
  - `kernel[(x, y, z,)](params…)`
- Calculate offsets
  - `tl.arange(0, N)[None, :] // H * stride`
- Access multi-dimensional tensors
  - `tl.load(offsets, masks, others)`

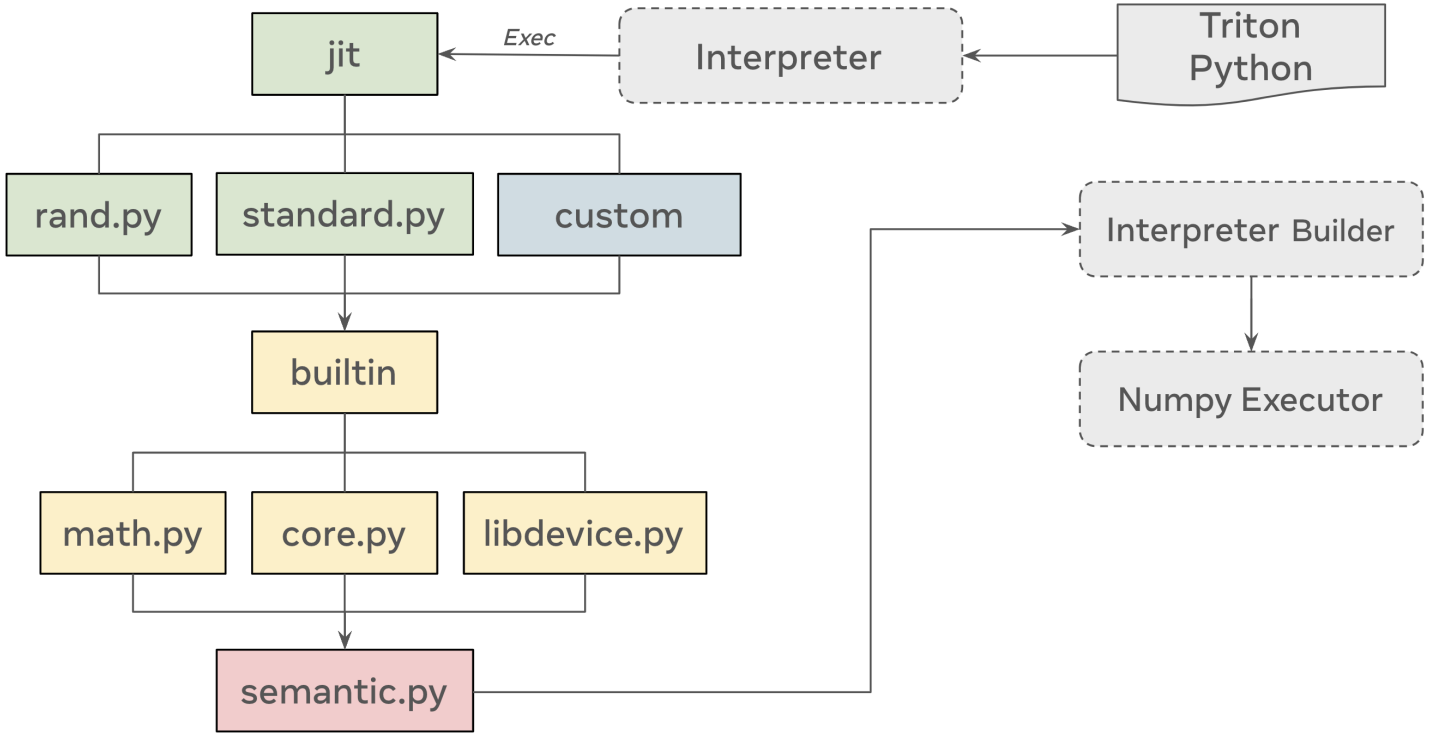
Triton Interpreter

- A debugger that allows users to debug Triton programs as if they were debugging standard Python programs on the CPU
    - Attach pdb to step through each statement interactively
    - Print tensor values as multidimensional arrays for better visualization
    - Serialize the execution of multiple Triton programs for easier debugging

Revisit the Frontend

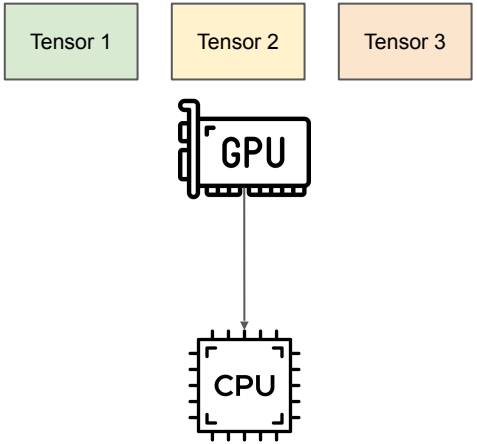


Frontend with the Interpreter

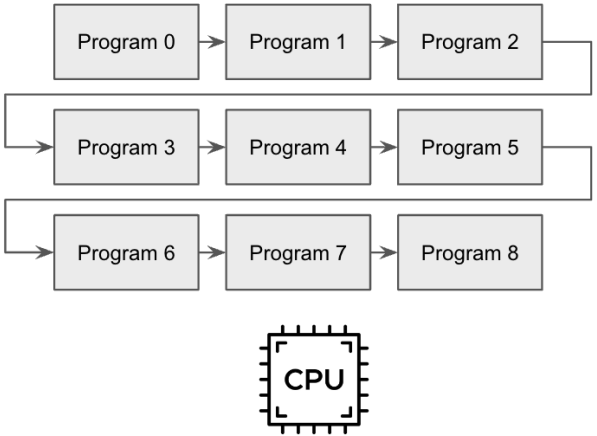


Interpreter Execution

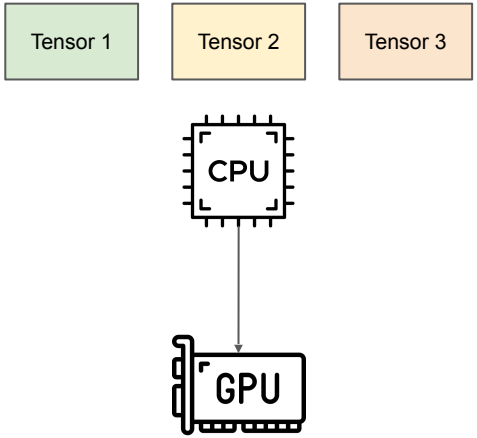
(1) Prepare CPU tensors



(2) Serialize execution



(3) Update GPU tensors



Demo 1: Print Tensors in the Multi-dim Array Format

```
z = x + y
```

vecAdd

```
@triton.jit
def add_kernel_device_print(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
  tl.device_print("offsets=", offsets)
  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)
```

Demo 1: Print Tensors in the Multi-dim Array Format

vecAdd

```
a = torch.randn(128, 128).cuda()
b = torch.randn(128, 128).cuda()
c = torch.zeros(128, 128).cuda()
add_kernel[(8, 8)](a, b, c, 128, 128, 16, 16)
```

```
pid (6, 5, 0) idx (10,  0) offsets=: 13648
pid (6, 5, 0) idx (10,  1) offsets=: 13649
pid (6, 5, 0) idx (10,  2) offsets=: 13650
pid (6, 5, 0) idx (10,  3) offsets=: 13651
pid (6, 5, 0) idx (10,  4) offsets=: 13652
pid (6, 5, 0) idx (10,  5) offsets=: 13653
pid (6, 5, 0) idx (10,  6) offsets=: 13654
pid (6, 5, 0) idx (10,  7) offsets=: 13655
pid (6, 5, 0) idx (10,  8) offsets=: 13656
pid (6, 5, 0) idx (10,  9) offsets=: 13657
pid (6, 5, 0) idx (10, 10) offsets=: 13658
pid (6, 5, 0) idx (10, 11) offsets=: 13659
```

Demo 1: Print Tensors in the Multi-dim Array Format

```
export TRITON_INTERPRET=1
print("offsets=", offsets)
```

```
offsets= [[   0    1    2    3    4    5    6    7    8    9   10   11   12   13
    14   15]
 [ 128  129  130  131  132  133  134  135  136  137  138  139  140  141
   142  143]
 [ 256  257  258  259  260  261  262  263  264  265  266  267  268  269
   270  271]
 [ 384  385  386  387  388  389  390  391  392  393  394  395  396  397
   398  399]
 [ 512  513  514  515  516  517  518  519  520  521  522  523  524  525
   526  527]
 [ 640  641  642  643  644  645  646  647  648  649  650  651  652  653
   654  655]
 [ 768  769  770  771  772  773  774  775  776  777  778  779  780  781
   782  783]
 [ 896  897  898  899  900  901  902  903  904  905  906  907  908  909
   910  911]
 [1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037
  1038 1039]
 [1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165
  1166 1167]
 [1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293
  1294 1295]
 [1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421
  1422 1423]
 [1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549
...
 [16240 16241 16242 16243 16244 16245 16246 16247 16248 16249 16250 16251
  16252 16253 16254 16255]
 [16368 16369 16370 16371 16372 16373 16374 16375 16376 16377 16378 16379
  16380 16381 16382 16383]]
```

Demo 2: Debug with PDB and the Interpreter

vecAdd

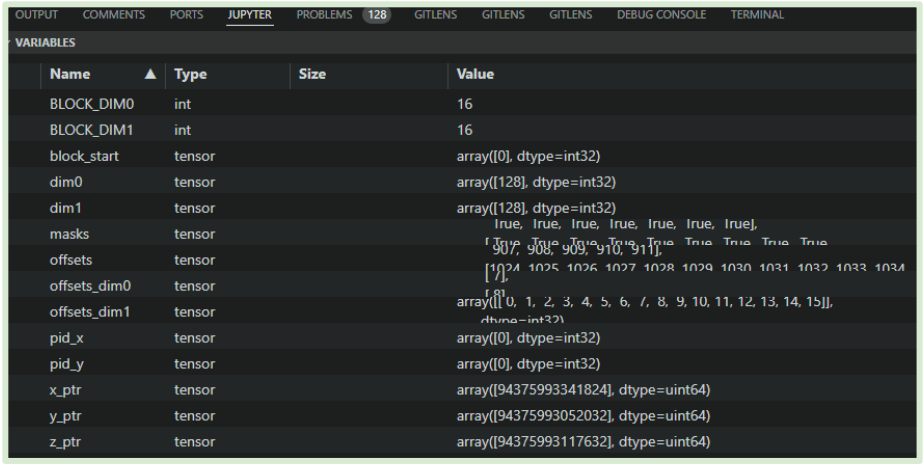
```
1  import pdb
2
3  @triton.jit
4  def add_kernel_pdb(x_ptr, y_ptr, z_ptr, dim0, dim1,
5                     BLOCK_DIM0: tl.constexpr, BLOCK_DIM1: tl.constexpr):
6    pid_x = tl.program_id(axis=0)
7    pid_y = tl.program_id(axis=1)
8    block_start = pid_x * BLOCK_DIM0 * dim1 + pid_y * BLOCK_DIM1
9    offsets_dim0 = tl.arange(0, BLOCK_DIM0)[:,None]
10   offsets_dim1 = tl.arange(0, BLOCK_DIM1)[None, :]
11   offsets = block_start + offsets_dim0 * dim1 + offsets_dim1
12   masks = (offsets_dim0 < dim0) & (offsets_dim1 < dim1)
13   if pid_x == 0 and pid_y == 0:
14     pdb.set_trace()
15   x = tl.load(x_ptr + offsets, mask=masks)
16   y = tl.load(y_ptr + offsets, mask=masks)
17   output = x + y
18   tl.store(z_ptr + offsets, output, mask=masks)
```

Demo 3: Debug with Jupyter Notebook

```python
import pdb

@triton.jit
def add_kernel_pdb(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)
```

Breakpoint



Variable View

Limitations of the Interpreter

- Precisions
  - Some floating-point types, like `float8` and `bfloat16`, support only a limited number of arithmetic operations
  - Hardware limitations also restrict full support in native Triton
    - e.g., float8 + float8 is not supported if no casting
- Concurrency bugs
  - Pairs of producers and consumers
- Indirect memory access
  - `ptr1 = tl.load(ptr0)`
  - `x = tl.load(ptr1)`

Outline

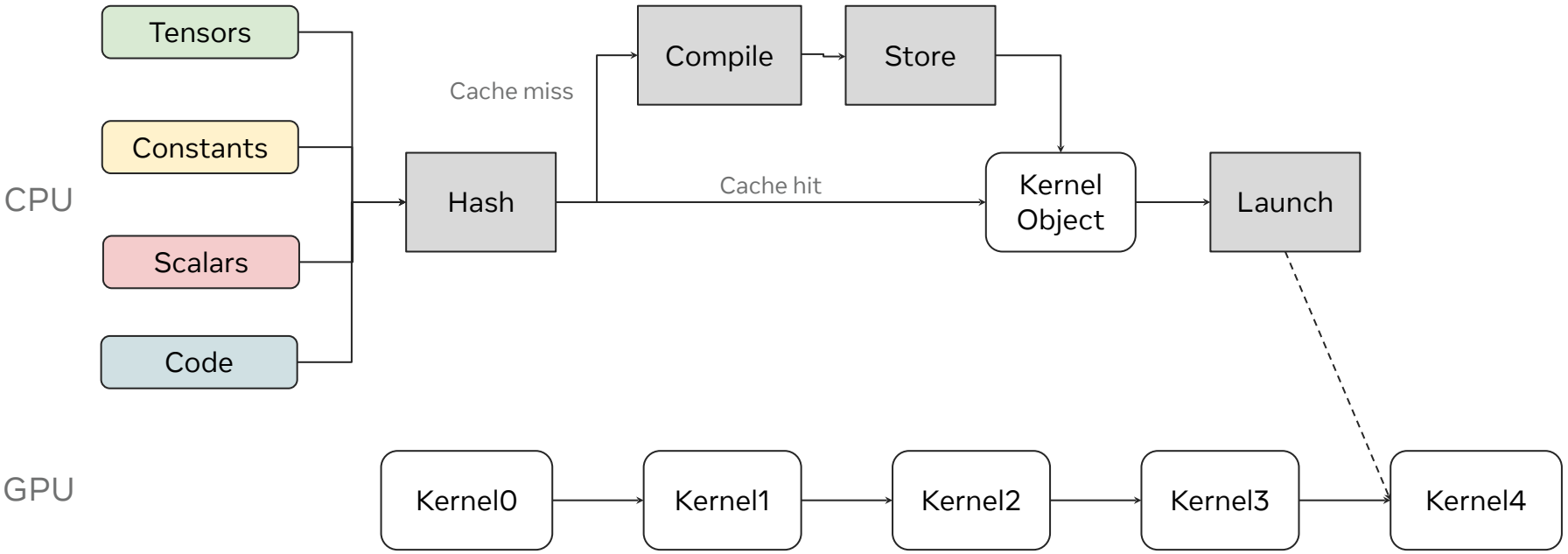
- Triton Review
- Interpreter
- Proton
- Proton: Instruction Sampling

Proton

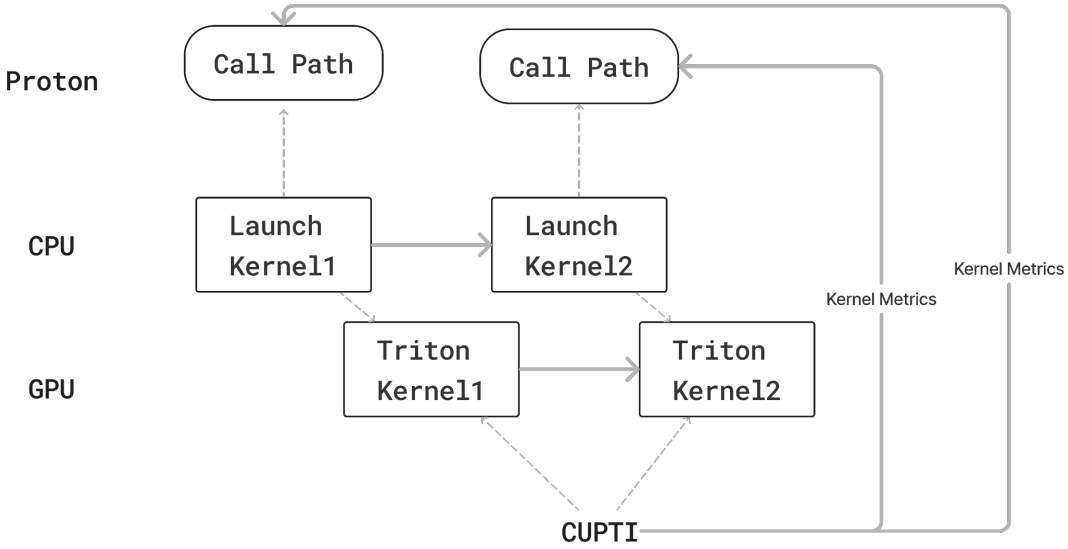
Proton (A **Pro**filer for Tri**ton**)

- 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

Triton Runtime



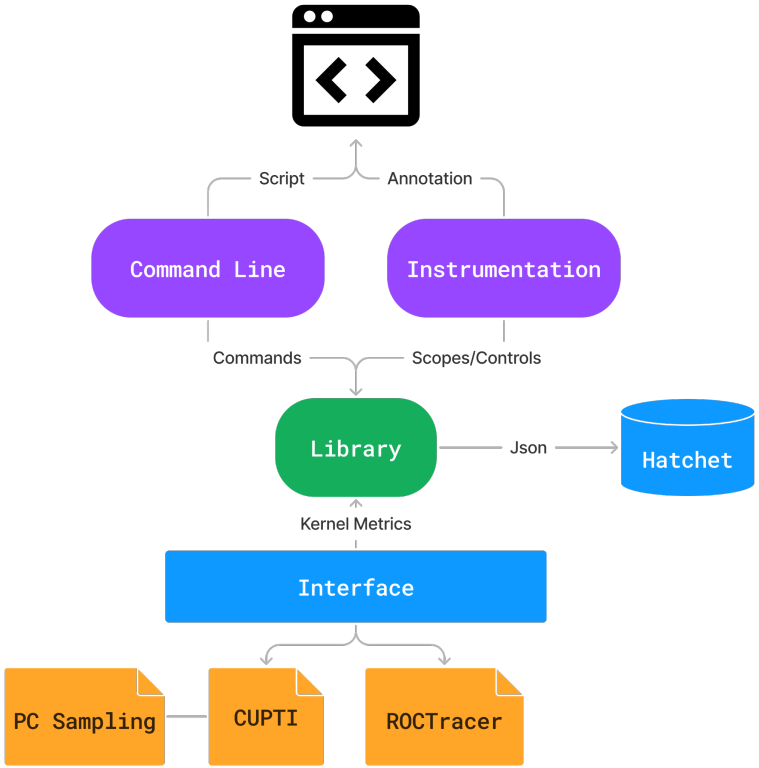
Proton Runtime



Aggregate timing into kernels with the same “group name”

sum/avg/min/max

Design


Script
Annotation
Command Line
Instrumentation
Commands
Scopes/Controls
Library
Json
Hatchet
Kernel Metrics
Interface
PC Sampling
CUPTI
ROCTracer

Call Path Profiling

- Profile kernel running time

```
55.193 ROOT
├─ 31.212 /home/kzhou6/gh200/triton/third_party/proton/tutorials/dynamic_net.py:<module>@98
|  └─ 31.212 /home/kzhou6/gh200/triton/python/triton/profiler/profile.py:wrapper@151
|     ├─ 0.002 /home/kzhou6/gh200/triton/third_party/proton/tutorials/dynamic_net.py:run@51
|     |  └─ 0.002 _ZN50_GLOBAL__N__c922cf59_17_RangeFactories_cu_38772b0829elementwise_kernel_with_indexI
_clEvENKUlvE0_clEvEUlIE_EEvT_T0_PN15function_traitsISD_E11result_typeE
|     ├─ 0.003 /home/kzhou6/gh200/triton/third_party/proton/tutorials/dynamic_net.py:run@52
|     |  └─ 0.003 _ZN2at6native29vectorized_elementwise_kernelILi4EZZZNS0_15sin_kernel_cudaERNS_18TensorIt
_T1_
|     ├─ 19.610 /home/kzhou6/gh200/triton/third_party/proton/tutorials/dynamic_net.py:run@66
|     |  └─ 19.610 /home/kzhou6/gh200/pytorch/torch/nn/modules/module.py:_wrapped_call_impl@1532
|     |     └─ 19.610 /home/kzhou6/gh200/pytorch/torch/nn/modules/module.py:_call_impl@1541
|     |        ├─ 13.931 /home/kzhou6/gh200/triton/third_party/proton/tutorials/dynamic_net.py:forward@36
|     |        |  ├─ 2.939 /home/kzhou6/gh200/pytorch/torch/_tensor.py:wrapped@40
|     |        |  |  ├─ 1.460 _ZN2at6native29vectorized_elementwise_kernelILi4EZNS0_53_GLOBAL__N__2ced54f0
18TensorIteratorBaseET0_EUlfE0_NS_6detail5ArrayIPcLi2EEEEEviS6_T1_
|     |        |  |  └─ 1.479 _ZN2at6native29vectorized_elementwise_kernelILi4EZNS0_53_GLOBAL__N__2ced54f0
18TensorIteratorBaseET0_EUlfE_NS_6detail5ArrayIPcLi2EEEEEviS6_T1_
|     |        |  ├─ 6.022 _ZN2at6native18elementwise_kernelILi128ELi2EZNS0_22gpu_kernel_impl_nocastINS0_1
eratorBaseERKT_EUliE_EEviT1_
|     |        |  ├─ 2.025 _ZN2at6native18elementwise_kernelILi128ELi2EZNS0_22gpu_kernel_impl_nocastINS0_1
|     |        |  └─ 2.945 _ZN2at6native29vectorized_elementwise_kernelILi4ENS0_15CUDAFunctor_addIfEENS_6
```

Python Context

```
54.763 ROOT
├─ 25.004 backward
|  ├─ 14.366 _ZN2at6native13reduce_kernelILi512ELi1ENS0_
|  ├─ 2.007 _ZN2at6native18elementwise_kernelILi128ELi2E
vEUlfffE_EEvRNS_18TensorIteratorBaseERKT_EUliE_EEviT1_
|  ├─ 2.461 _ZN2at6native29vectorized_elementwise_kernel
|  ├─ 5.725 _ZN2at6native29vectorized_elementwise_kernel
|  └─ 0.446 _ZN2at6native29vectorized_elementwise_kernel
├─ 19.399 forward
|  ├─ 7.961 _ZN2at6native18elementwise_kernelILi128ELi2E
EUliE_EEviT1_
|  ├─ 2.018 _ZN2at6native18elementwise_kernelILi128ELi2E
|  ├─ 4.415 _ZN2at6native29vectorized_elementwise_kernel
|  ├─ 1.455 _ZN2at6native29vectorized_elementwise_kernel
seET0_EUlfE0_NS_6detail5ArrayIPcLi2EEEEEviS6_T1_
|  ├─ 2.073 _ZN2at6native29vectorized_elementwise_kernel
seET0_EUlfE2_NS_6detail5ArrayIPcLi2EEEEEviS6_T1_
|  └─ 1.477 _ZN2at6native29vectorized_elementwise_kernel
seET0_EUlfE_NS_6detail5ArrayIPcLi2EEEEEviS6_T1_
├─ 0.004 init
|  ├─ 0.003 _ZN2at6native29vectorized_elementwise_kernel
|  └─ 0.001 _ZN50_GLOBAL__N__c922cf59_17_RangeFactories_
NKUlvE0_clEvEUlIE_EEvT_T0_PN15function_traitsISD_E11resu
├─ 4.412 loss
|  ├─ 2.949 _ZN2at6native13reduce_kernelILi512ELi1ENS0_8
|  └─ 1.462 _ZN2at6native29vectorized_elementwise_kernel
```

Shadow Context

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

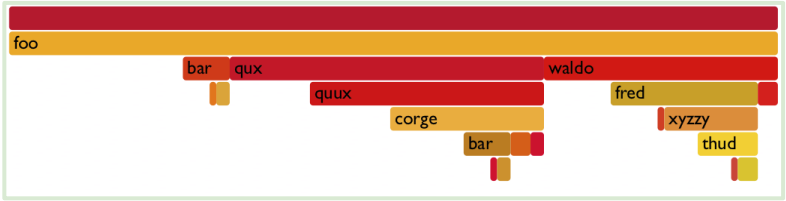
- Lightweight source code instrumentation
  - Profile start/stop/finalize
  - Scopes
  - Hooks
- Command line
  - `python -m proton main.py`
  - `proton main.py`

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)`

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



Demo 4: Instrumentation-based Profiling

- The default profile name is `proton.hatchet`

vecAdd

```
1  proton.start()
2  a = torch.randn(128, 128, device='cuda')
3  b = torch.randn(128, 128, device='cuda')
4  c = torch.zeros(128, 128, device='cuda')
5  add_kernel[(8, 8)](a, b, c, 128, 128, 16, 16)
6  proton.finalize()
7  c_ref = a + b
8  assert torch.allclose(c, c_ref)
```

Demo 4: Instrumentation-based Profiling

- `proton-viewer -m time/ms ./proton.hatchet`

```
0.014 ROOT
├─ 0.002 _ZN2at6native29vectorized_elementwise_kern
├─ 0.010 _ZN2at6native54_GLOBAL__N__d8ceb000_21_Dis
└─ 0.002 add_kernel

Legend (Metric: time/ms (inc) Min: 0.00 Max: 0.01)
  0.01 - 0.01
  0.01 - 0.01
  0.01 - 0.01
  0.01 - 0.01
  0.00 - 0.01
  0.00 - 0.00
```

Demo 5: Activate/Deactivate Proton

- What if I only want to profile Triton kernels?
  - deactivate/activate allows you to select interesting regions to profile

vecAdd

```python
session_id = proton.start()
proton.deactivate(session_id)
a = torch.randn(128, 128, device='cuda')
b = torch.randn(128, 128, device='cuda')
c = torch.zeros(128, 128, device='cuda')
proton.activate(session_id)
add_kernel[(8, 8)](a, b, c, 128, 128, 16, 16)
proton.finalize()
c_ref = a + b
assert torch.allclose(c, c_ref)
```

```
0.009 ROOT
└─ 0.009 add_kernel

Legend (Metric: time/ms (inc) Min: 0.01 Max: 0.01)
0.01 - 0.01
0.01 - 0.01
0.01 - 0.01
0.01 - 0.01
0.01 - 0.01
0.01 - 0.01
```

Scopes

- A user-defined region with semantic information
  - Initialization
  - Forward
  - Backward

- `with proton.scope(name)`

Demo 6: Profiling Using Scopes

- `proton-viewer -m time/ms ./proton.hatchet`

vecAdd

```
1  def run():
2    with proton.scope("initialization"):
3      a = torch.randn(128, 128, device='cuda')
4      b = torch.randn(128, 128, device='cuda')
5      c = torch.zeros(128, 128, device='cuda')
6    with proton.scope("add"):
7      add_kernel[(8, 8)](a, b, c, 128, 128, 16, 16)
```

```
0.639 ROOT
├─ 0.090 add
│  └─ 0.090 add_kernel
└─ 0.549 initialization
   ├─ 0.069 _ZN2at6native29vectorized_elementwise_
   └─ 0.480 _ZN2at6native54_GLOBAL__N__d8ceb000_21

Legend (Metric: time/ms (inc) Min: 0.07 Max: 0.64)
0.58 - 0.64
0.47 - 0.58
0.35 - 0.47
0.24 - 0.35
0.13 - 0.24
0.07 - 0.13
```

Metrics

- Hardware metrics
  - Come from profiling substrates (e.g., CUPTI)
    - Kernel time
    - Instruction samples

- User-defined metrics
  - Come from users
    - Flops
    - Bytes
    - Tokens

Demo 7: Profiling with Metrics

- How many bytes many been accessed in each scope?
  - `proton.scope(name, {“metric-name”: metric-value})`

vecAdd

```python
def run():
  with proton.scope("initialization", metrics={"bytes": 128*128*4*3}):
    a = torch.randn(128, 128, device='cuda')
    b = torch.randn(128, 128, device='cuda')
    c = torch.zeros(128, 128, device='cuda')
  with proton.scope("add", metrics={"bytes": 128*128*4*3}):
    add_kernel[(8, 8)](a, b, c, 128, 128, 16, 16)

for _ in range(10):
  run()
```

```
6.157 3932160.000 ROOT
├─ 22.037 1966080.000 add
└─ 3.578 1966080.000 initialization

Legend (Metric: gbyte/s (inc) Min: 3.58 Max: 22.04)
20.19 - 22.04
16.50 - 20.19
12.81 - 16.50
9.12 - 12.81
5.42 - 9.12
3.58 - 5.42
```

Rename Triton Kernels

- How can Triton kernels be differentiated based on different block sizes?

vecAdd

```
@triton.jit
def matmul_kernel(
    a_ptr, b_ptr, c_ptr,
    M, N, K,
    stride_am, stride_ak,
    stride_bk, stride_bn,
    stride_cm, stride_cn,
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
    GROUP_SIZE_M: tl.constexpr,
    ACTIVATION: tl.constexpr,
):
```

Rename Triton Kernels

- How do you annotate the grid sizes in a Triton program?
- How do you specify the number of warps used in a Triton kernel?
- We want to rename a Triton function with a custom name
    - `matmul_<num_warps:4>_<block_m:16>_<grid_1x1x2>`

Metrics Association

- How to associate each Triton kernel with metrics based on kernel arguments?
  - flops{8, 16, 32, 64}
  - bytes
- Using scope if OK but need to instrument the script at a lot if there are many kernel launches

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

Demo 8: Matmul

- `proton/tutorials/matmul.py`
- `proton-viewer -m time/ms -i ".*triton.*" ./matmul.hatchet`

vecAdd

```python
def metadata_fn(
  grid: tuple,
  metadata: NamedTuple,
  args: dict,
):
  grid_x, grid_y, grid_z = unpack_grid(grid)
  num_warps = metadata.num_warps
  num_stages = metadata.num_stages
  cluster_x, cluster_y, cluster_z = metadata.cluster_dims
  shared_memory = metadata.shared
  M, K = args["a_ptr"].shape
  K, N = args["b_ptr"].shape
  return {
    "name":
    f"matmul_<grid:{grid_x}x{grid_y}x{grid_z}>_"
    "<cluster:{cluster_x}x{cluster_y}x{cluster_z}>_"
    "<warps:{num_warps}>_<shared:{shared_memory}>_"
    "<stages:{num_stages}>",
    "flops": 2 * M * N * K,
    "bytes": (M * N + N * K + K * M) * args["a_ptr"].element_size(),
  }
```

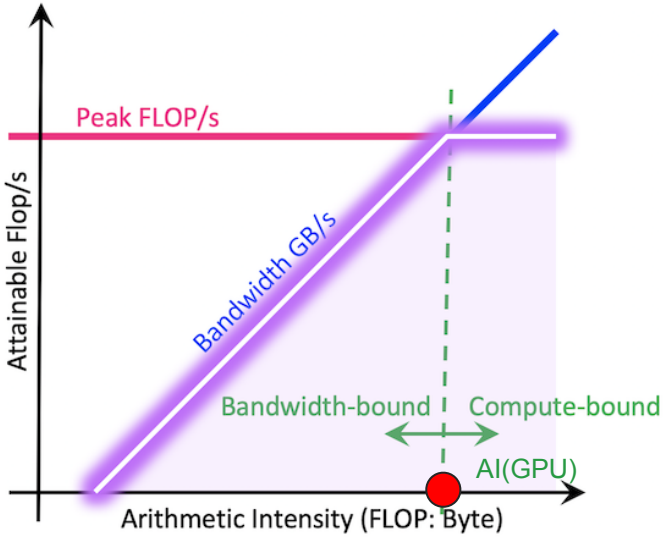
Demo 8: Matmul

- We use scopes to annotate
  - Matmul shapes: matmul_M_N_K
  - Autotuned configurations: <autotune>

```
├─ 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>
```

Roofline-based Utilization

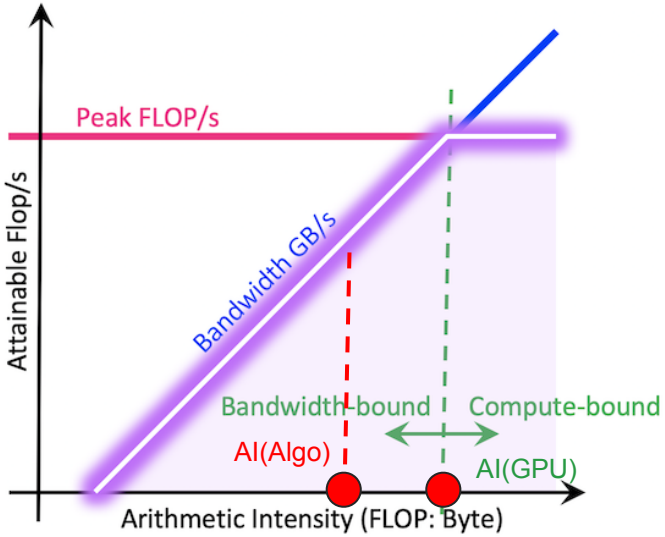
- Arithmetic Intensity (AI) of the GPU
  - AI(GPU) = maximum_flops / maximum_dram_bandwidth
  - maximum_flops can refer to fp8, fp8_tensor, fp16, fp16_tensor, fp32, tf32, fp64



Roofline Performance Model - NERSC Documentation

Roofline-based Utilization

- Arithmetic Intensity (AI) of the algorithm
  - AI(Algo) = expected_flops / expected_bytes



Roofline Performance Model - NERSC Documentation

Roofline-based Utilization

- Utilization
  - Util = flop/s / maximum_flop/s
  - flop/s = expected_flops / time/s
  - maximum_flop/s = min(AI(Algo) / AI(GPU), 1.0) * maximum_flops

Demo 8: Matmul Roofline Utilization

- `proton-viewer -m util,tflop/s -i ".*triton.*" ./matmul.hatchet`

```
├─ 0.306 18.367 matmul_<grid:128x1x1>_<cluster:1x1x1>_<warps:4>_<shared:36864>_<stages:4>
├─ 0.193 11.590 matmul_<grid:256x1x1>_<cluster:1x1x1>_<warps:4>_<shared:30720>_<stages:4>
├─ 0.324 19.458 matmul_<grid:32x1x1>_<cluster:1x1x1>_<warps:8>_<shared:98304>_<stages:3>
├─ 0.166 9.973 matmul_<grid:512x1x1>_<cluster:1x1x1>_<warps:2>_<shared:24576>_<stages:5>
├─ 0.342 20.534 matmul_<grid:64x1x1>_<cluster:1x1x1>_<warps:4>_<shared:49152>_<stages:4>
└─ 0.310 18.652 matmul_<grid:64x1x1>_<cluster:1x1x1>_<warps:4>_<shared:61440>_<stages:4>
```

Outline

- Triton Review
- Interpreter
- Proton
- Proton: Instruction Sampling

Proton:
Instruction Sampling

Background

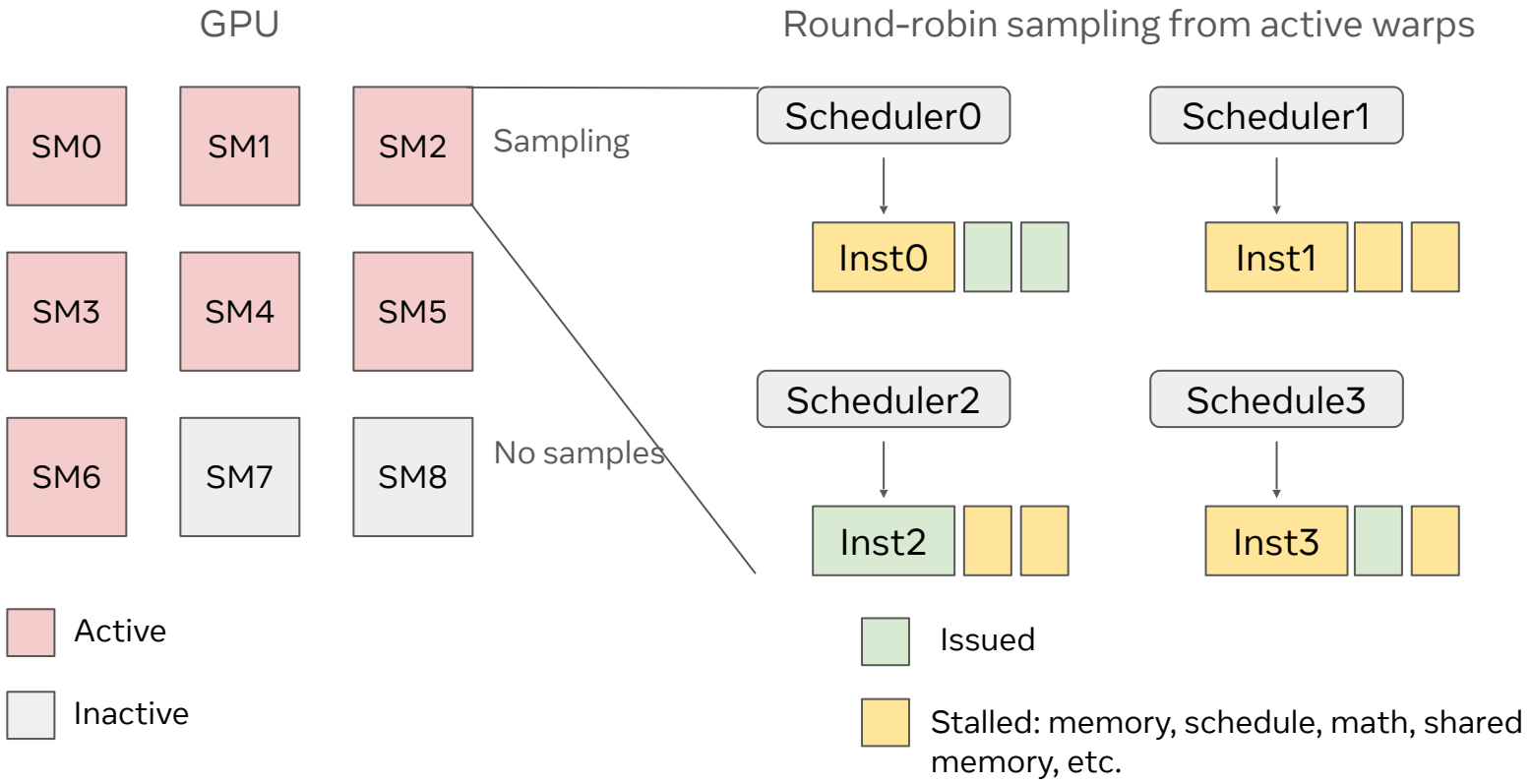
- For large functions, we need fine-grained insights about which lines/IRs/instructions are expensive

- Instruction sampling is an experimental feature we're developing to support this goal
    - It's called pc sampling using NVIDIA's terminology

Instruction sampling

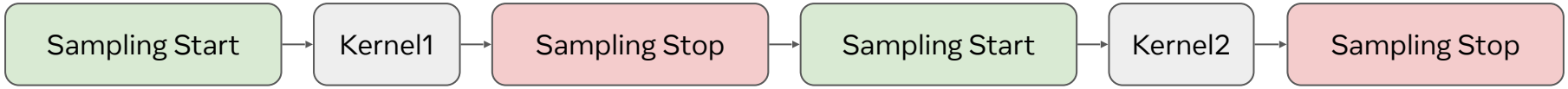
- Sample an instruction on each active GPU SM every N cycles
- Each instruction is associated with a stall reason if available
    - Why the instruction was not issued
- “Low overhead” with regard to each kernel’s GPU time
- Available on NVIDIA, AMD and Intel GPUs

Instruction Sampling on NVIDIA GPUs



Proton Implementation

- Serialized: start PC sampling before/after each kernel launch
  - Simple and reliable but relative higher overhead



- Continuous: start PC sampling before/after each region
  - Low overhead but needs auxiliary data structures to attribute metrics
    - The same kernel can be invoked at two different contexts

Overhead Comparison with Nsight Compute

- https://pytorch.org/tutorials//beginner/examples_nn/dynamic_net.html
  - A simple neural network with many small kernels
- NCU
  - SourceCounters
    - Easy to use but significant overhead
      - `ncu --section SourceCounters python ./dynamic_net.py`
        - 4s-> 66 mins (>1000x)
  - PM Sampling
    - Difficult to use and still high overhead
      - `ncu --metrics smsp__warps_issue_stalled_wait python dynamic_net.py`
        - 4s->16 mins (~250x)

Overhead Comparison with Nsight Compute

- Proton
  - Easy to use and relatively low overhead
    - `proton --backend cupti_pcsampling dynamic_net.py`
  - Serialized mode: 20x
  - Can be reduced to less than 5x with the continuous mode

Demo 9: Hotspots within Matmul Kernels

- `proton-viewer -m num_samples,num_stalled_samples -i ".*matmul_1920_1920_1920.*" -t 1000 ./matmul.hatchet`
  - -i: Only includes call paths passing through this matched frame
  - -t: Only includes call paths with a value greater than the given threshold

vecAdd

```
1 proton.start("matmul", backend="cupti_pcsampling", hook="triton")
```

Demo 9: Hotspots within Matmul Kernels Kernels

```
1851039.000 1015711.000 ROOT
└─ 1851039.000 1015711.000 matmul_1920_1920_1920
   ├─ 699125.000 377785.000 _ZN2at6native29vectorized_elementwise_kernelILi4ENS0_11FillFunctorIiEENS_6detail5ArrayI
   ├─ 230100.000 127721.000 cublas
   │  └─ 230100.000 127721.000 sm90_xmma_gemm_f16f16_f16f32_f32_nn_n_tilesize128x128x64_warpgroupsize1x1x1_execute_
   └─ 921814.000 510205.000 triton
      ├─ 734034.000 396343.000 _ZN2at6native29vectorized_elementwise_kernelILi4ENS0_11FillFunctorIiEENS_6detail5Arr
      └─ 187780.000 113862.000 matmul_<grid:120x1x1>_<cluster:1x1x1>_<warps:8>_<shared:147456>_<stages:3>_<tiles:12
         ├─ 38080.000 22653.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@110
         ├─ 52796.000 28997.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@113
         ├─ 38663.000 27270.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@114
         ├─ 37325.000 22239.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@116
         ├─ 5195.000 3975.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@119
         ├─ 1299.000 782.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@124
         ├─ 2964.000 1776.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@130
         └─ 11458.000 6170.000 /home/kzhou6/gh200/triton/third_party/proton/tutorials/matmul.py:matmul_kernel@132
```

```
109     accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
110     for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
111         # Load the next block of A and B, generate a mask by checking the K dimension.
112         # If it is out of bounds, set it to 0.
113         a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
114         b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
115         # We accumulate along the K dimension.
116         accumulator += tl.dot(a, b)
117         # Advance the ptrs to the next K block.
118         a_ptrs += BLOCK_SIZE_K * stride_ak
119         b_ptrs += BLOCK_SIZE_K * stride_bk
```

Wrap Up

Code

- All codes are available on Github
  - https://tinyurl.com/3uxsd5up
- Option 1: install using pip wheel
  - Not all features are available
- Option 2: install from source
  - May take a while (~10 mins on an M3 macbook)

Caution

- Not all examples can be run in Jupyter notebook

- You may encounter errors such as
  - RuntimeError: Could not find `libcupti.so`. Make sure it is in your `LD_LIBRARY_PATH`
    - The issue has been resolved in recent Triton updates
    - Install triton/main from source
  - Permission issue with Performance Counters
    - Run the tool or application with elevated privileges or enable access for all users.

Ongoing Work

- Performance analyzer
  - Instrument compiler analyses/passes to emit performance warnings
  - Associate static performance warnings with dynamic profile to provide guidance
    - e.g., kernel.py: 9 -> warning: vectorization fails

- Fine-grained performance metrics
  - Collect time and other fine-grained metrics through LLVM/MLIR instrumentation

Acknowledgements

- OpenAI
  - Supported the design and implementation of Proton
- AMD
  - Led the development of the ROCTracer backend
- Meta
  - Provided qualitative feedback on tool usage and performance
- NVIDIA
  - Offered advice and bug fixes for the CUPTI PC sampling API

Takeaways

- Use Triton Interpreter to debug tensor/variable values
- Use Proton for fast feedback, lightweight profiling, and more insights about
  - Calling context
  - Flop/s and utilization
  - Kernel specialization

Thank you!

Triton Conference 2024