

Practical Performance Optimization for Deep Learning Applications

Keren Zhou & Philippe Tillet

keren.zhou@rice.edu

phil@openai.com

GPUs are Underutilized

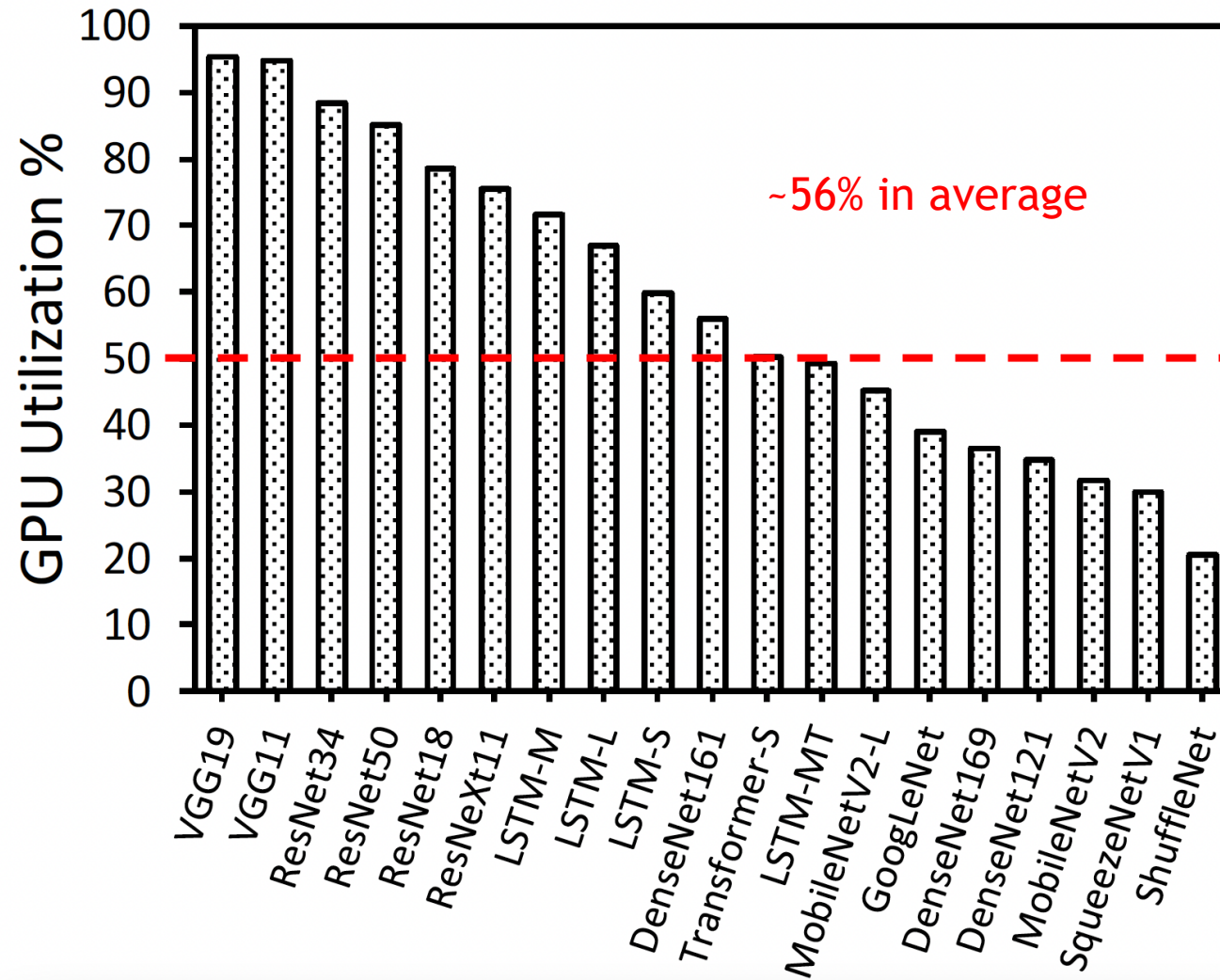
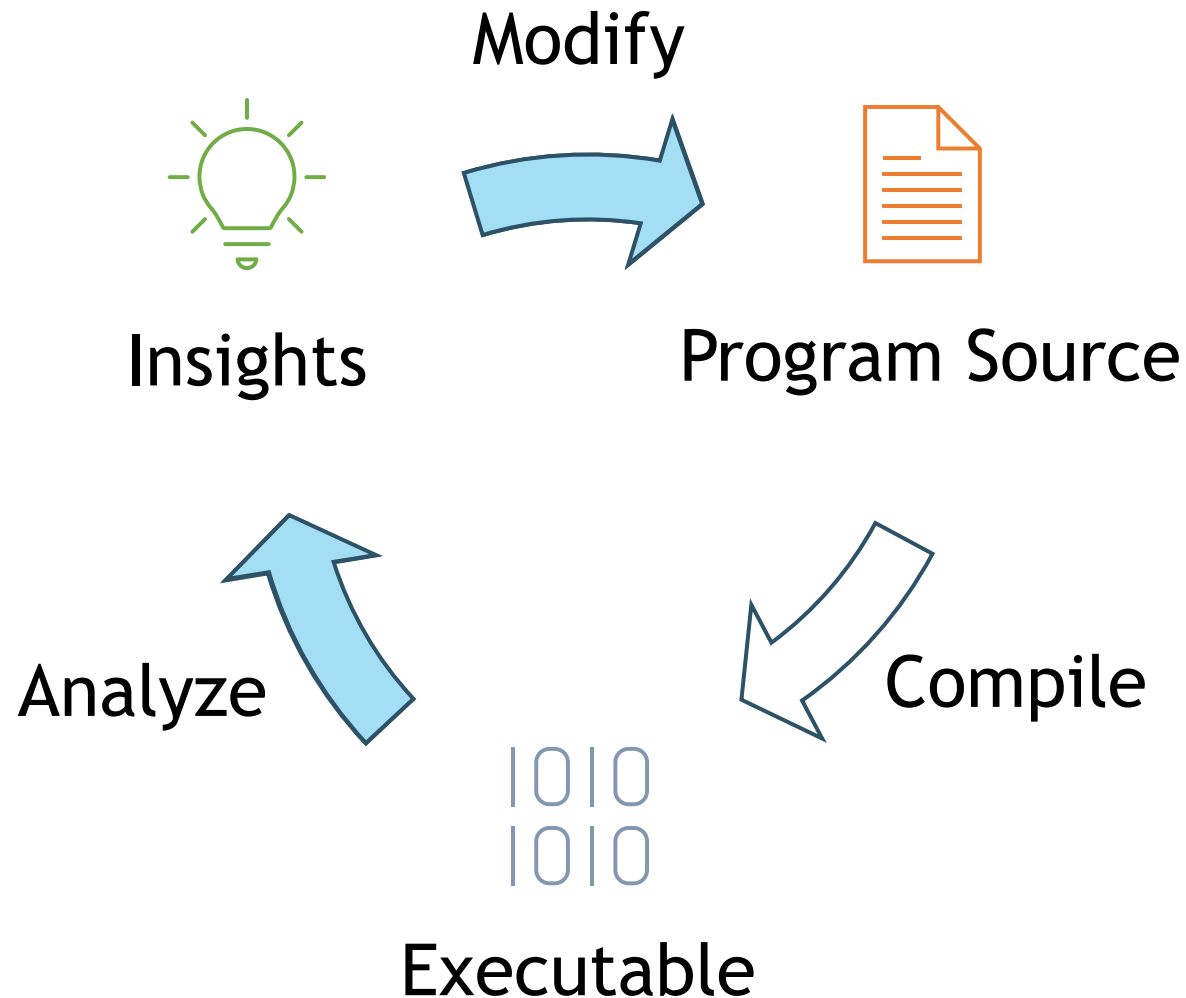


Image source:

Yeung, Gingfung, et al. "Towards GPU utilization prediction for cloud deep learning." 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20). 2020.

GPU Program Optimization Process



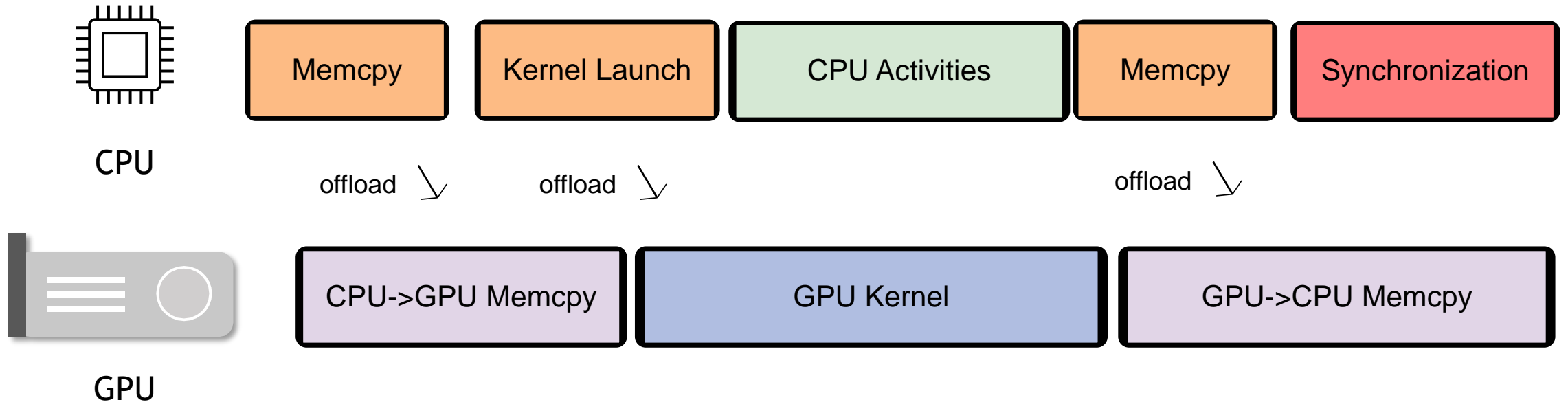
Optimization Techniques

- What will not be covered today
 - Quantization
 - Compression
 - Pruning
 - Sparse computation
- What will be covered today

Given a GPU kernel, how do you optimize its implementation?

Given a PyTorch script, how do you pinpoint its performance bottlenecks?

GPU-accelerated Application Sketch



GPU-accelerated Application Sketch

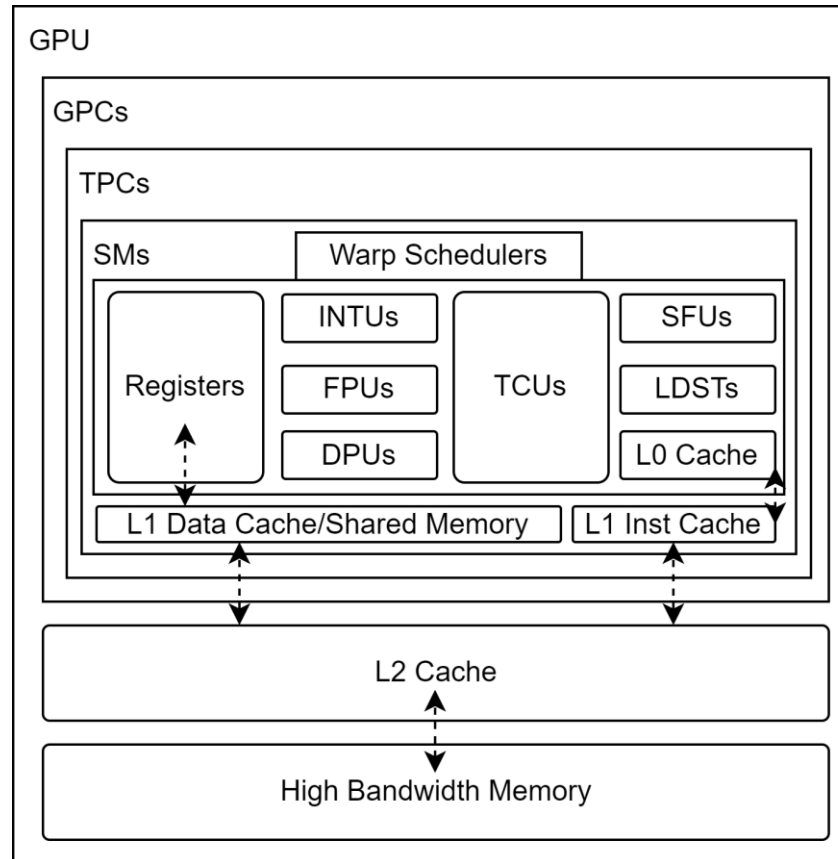


GPU

CPU->GPU Memcpy

GPU Kernel

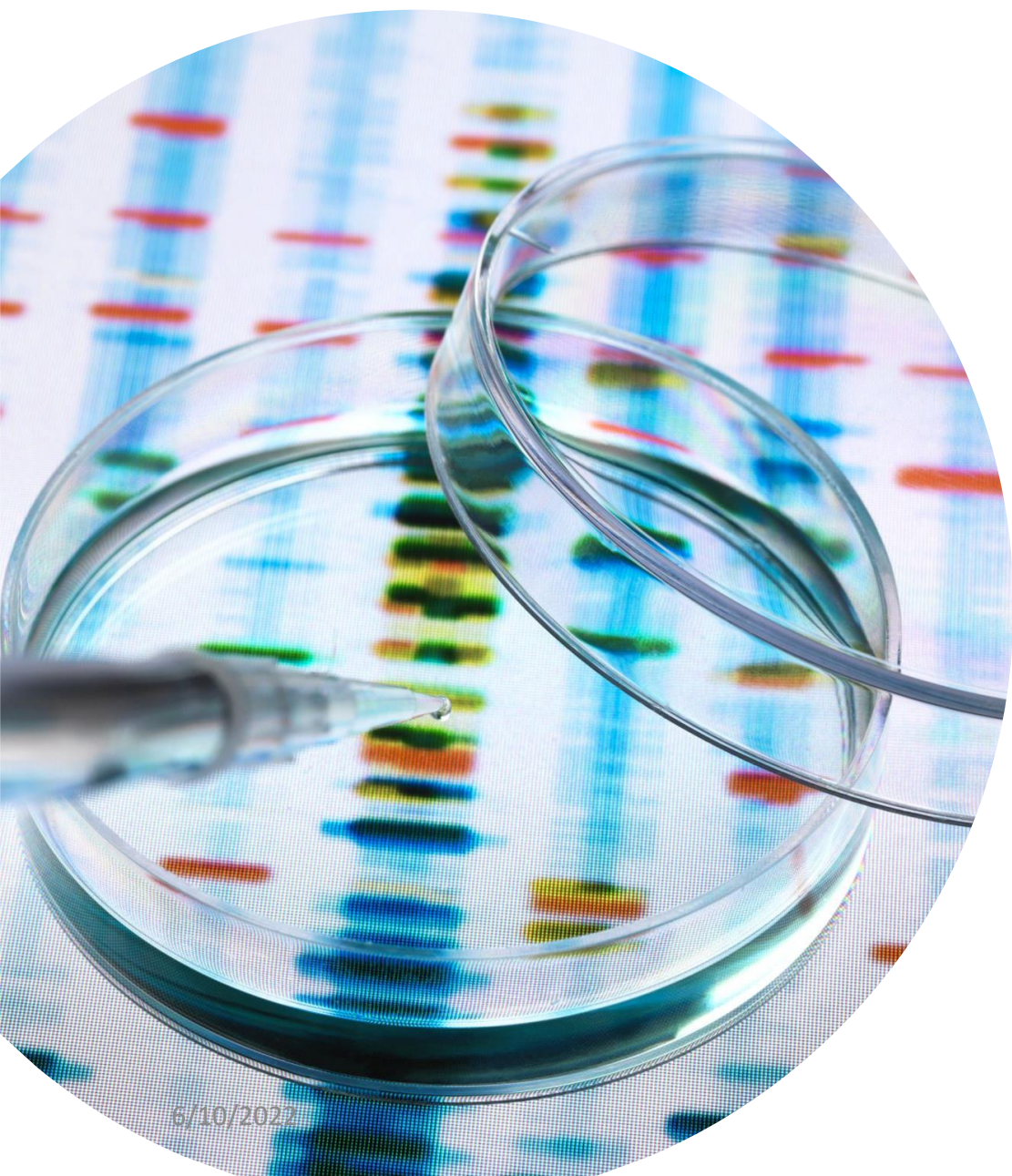
GPU->CPU Memcpy



NVIDIA GA100 architecture

Performance Analysis and Optimization for GPU Code is Challenging

- Sophisticated programming models
 - Tensorflow, PyTorch, RAJA, and Kokkos
- Complicated hardware
 - Multiple compute units
 - Multiple memory spaces
 - Thread synchronization/divergence
- Frequent communication
 - Data transfers between GPUs and CPUs
 - Internode communication



GPU Kernel Optimization

Inefficiencies of Existing PyTorch Operators

- Native PyTorch operators (e.g., `torch.add`)
 - Can be very slow
 - Can run out-of-memory
 - Graph compilers (e.g., TorchScript)
 - Don't support custom data-structures
 - block-sparse tensors
 - lists/trees of tensors
 - Don't support custom precision format
 - FP8
 - Automatic kernel fusion is limited
- Customize GPU kernel implementation!

How Difficult It Is to Optimize a GEMM Kernel?

- Vanilla (1-10% fp32 peak)
- NVIDIA CUDA Programming Guide (30%-50% fp32 peak)
 - +global memory coalesce
 - +shared memory

C/C++

- CUTLASS (80%-90% tf32 peak)
 - +tf32 tensor core
 - +vectorization
 - +shared bank conflict reduction
 - +thread layout autotune
 - +async shared memory transfer
 - +multi-stage shared memory

C++ Template & PTX

- cuBLAS (>90% tf32 peak)
 - +register bank conflict reduction
 - +control code optimization

SASS

Difficulty



Problems with Handwritten GPU Kernels

- Hard to recruit new Machine Learning Engineers
- Difficult to maintain libraries in a small company
- A black box to Machine Learning researchers
 - They want to understand how kernels work
 - They want to fast validate new ideas at scale

Triton - Agile Development of Fast GPU Kernels

- PyTorch compatible
 - Tensors are stored on-chip rather than off-chip
 - Custom data-structures using tensors of pointers
 - Python syntax
 - All standard python control flow structure (for/if/while) are supported
 - Highly optimized GPU code is generated
 - +tf32 tensor core
 - +vectorization
 - +shared bank conflict reduction
 - +thread layout autotune
 - +async shared memory transfer
 - +multi-stage shared memory
- } Automatic apply with minimal annotations

Triton vs CUDA

	CUDA	Triton
Memory	Global/Shared/Local	Automatic
Parallelism	Threads/Blocks/Warps	Mostly Blocks
Tensor Core	Manual	Automatic
Vectorization	.8/.16/.32/.64/.128	Automatic
Async SMT	Support	Limited
Device Function	Support	Not Available

Vector Addition

- $Z[:] = X[:] + Y[:]$
 - Without boundary check
- `@triton.jit`
 - Kernel decorator
- `tl.load()`
 - Load values from global memory to shared memory/registers
- `_add[grid](num_warps= K)`
- `grid = (G,)`
 - G thread block
- `num_warps = K`
 - $K = 4$ by default

```
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, 1024)
    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets

    # load 1024 elements of X, Y, Z
    x = tl.load(x_ptrs)
    y = tl.load(y_ptrs)

    # do computations
    z = x + y

    # write-back 1024 elements of X, Y, Z
    tl.store(z_ptrs, z)

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (1, )
_add[grid](z, x, y, N)
```

Vector Addition - Boundary Check

- $Z[:] = X[:] + Y[:]$
 - With boundary check
- `program_id()`
 - Get the block id
- `mask`
 - if `mask[idx]` is false, do not load the data at address `pointer[idx]`
- `triton.cdiv(N, 1024)`
 - $(N - 1) // 1024 + 1$

```
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, 1024)
    offsets += tl.program_id(0)*1024
    # create pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

Vector Addition - Custom Tile Size

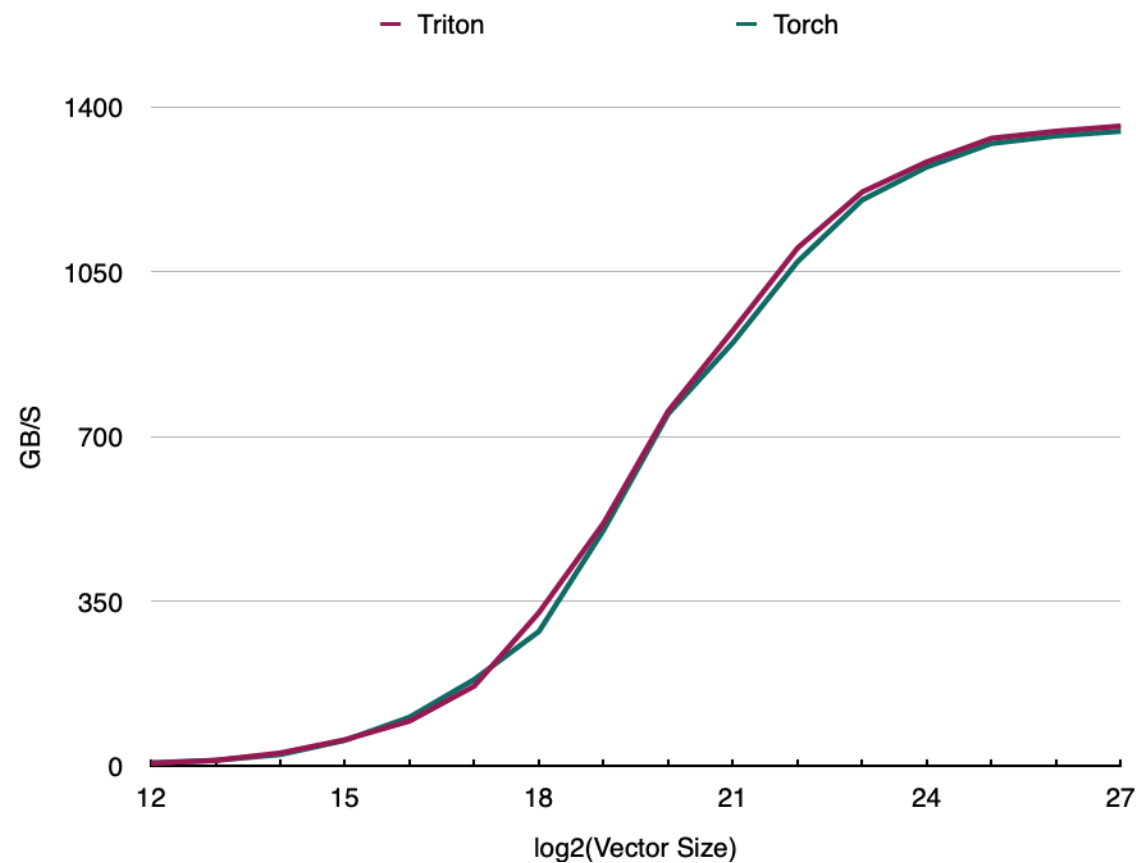
- $Z[:] = X[:] + Y[:]$
 - Each block computes *TILE* elements
- @triton.autotune
 - Instantiate kernels using configs
 - Select the best config based on the execution time
- lambda
 - Calculate grid dim based on *TILE*

```
import triton.language as tl
import triton
@triton.autotune(configs=
    [triton.Config('TILE': 128),
     triton.Config('TILE': 256)]
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N, TILE: tl.constexpr):
    # same as torch.arange
    offsets = tl.arange(0, TILE)
    offsets += tl.program_id(0)*TILE
    # create pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = lambda args: (triton.cdiv(N, args["TILE"]), )
_add[grid](z, x, y, N)
```

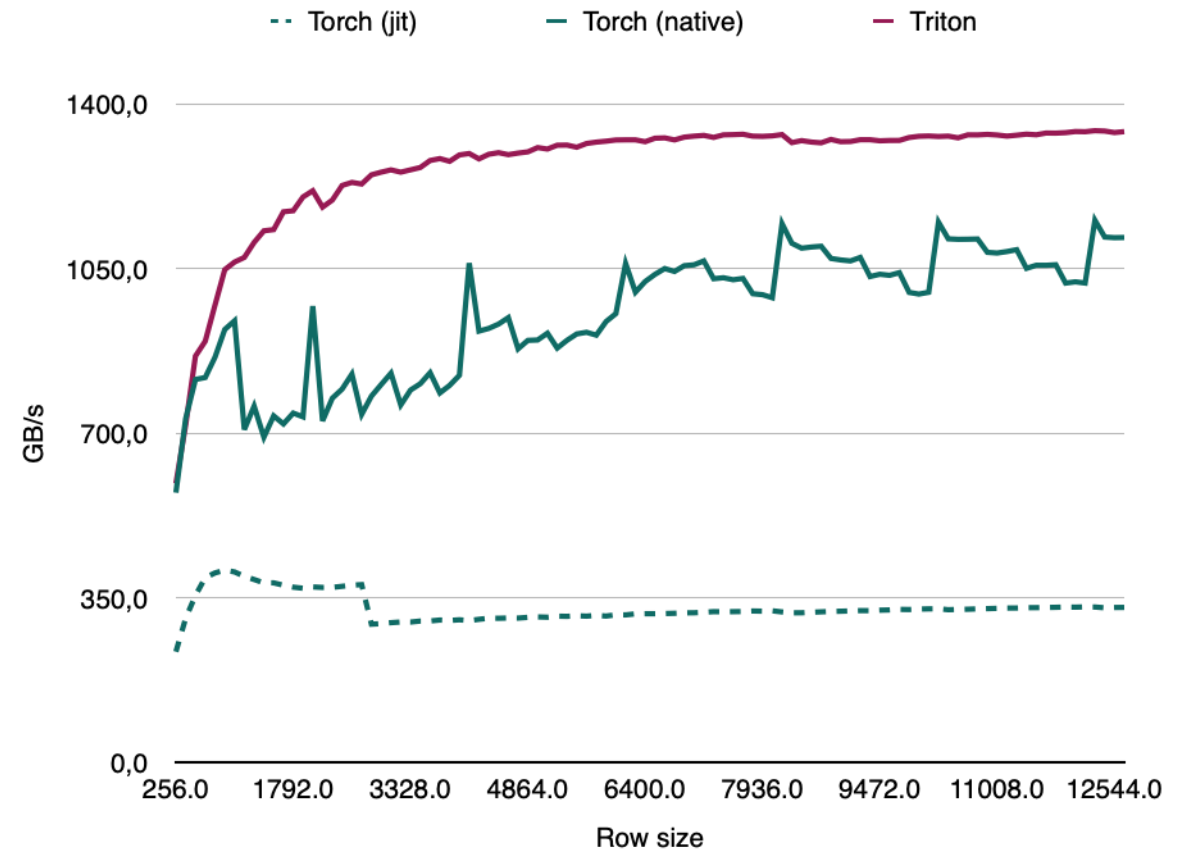

Element-wise OP Performance

- Triton and Torch both achieve peak bandwidth
- Researchers can write *fused element-wise* operations easily using Triton



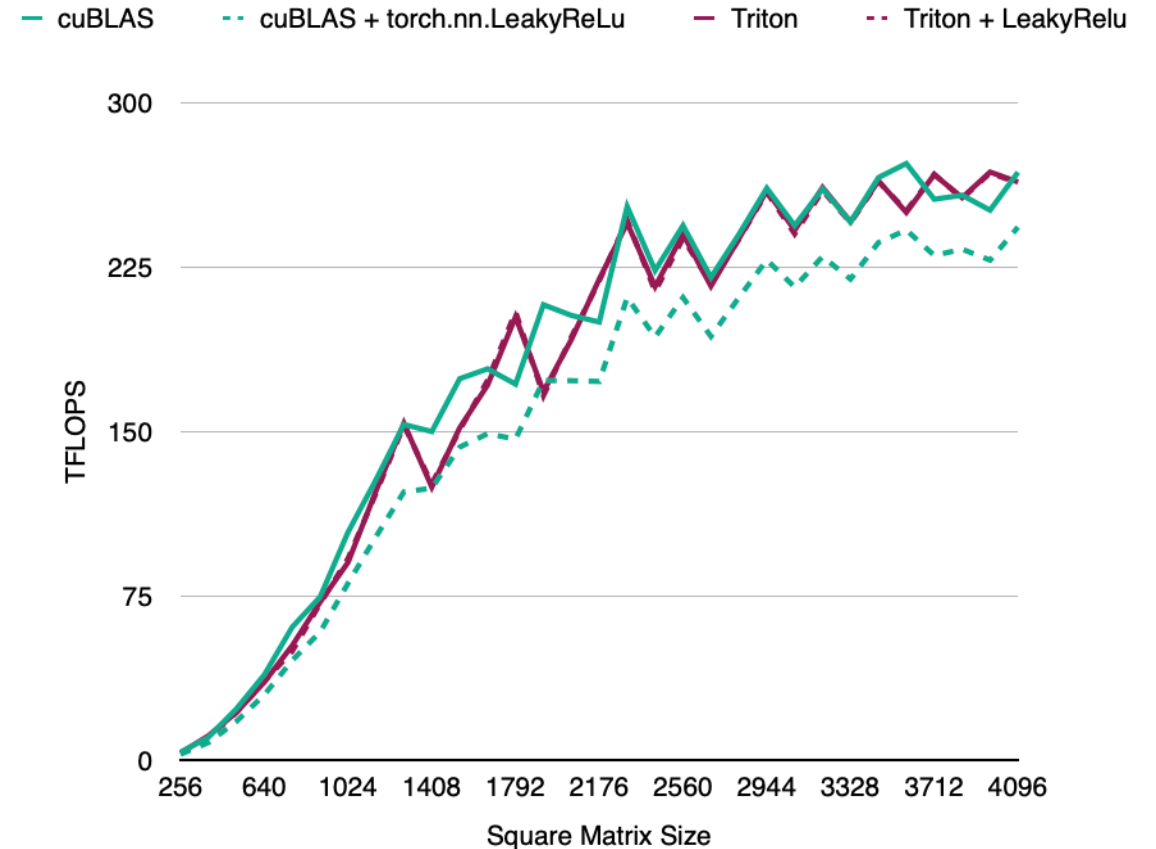
Row-wise Normalization Performance

- Triton kernels can keep data on-chip throughout the entire normalization
- PyTorch JIT could in theory do that but in practice doesn't
- The native PyTorch op is designed to work for every input shape and is slower in cases where we care



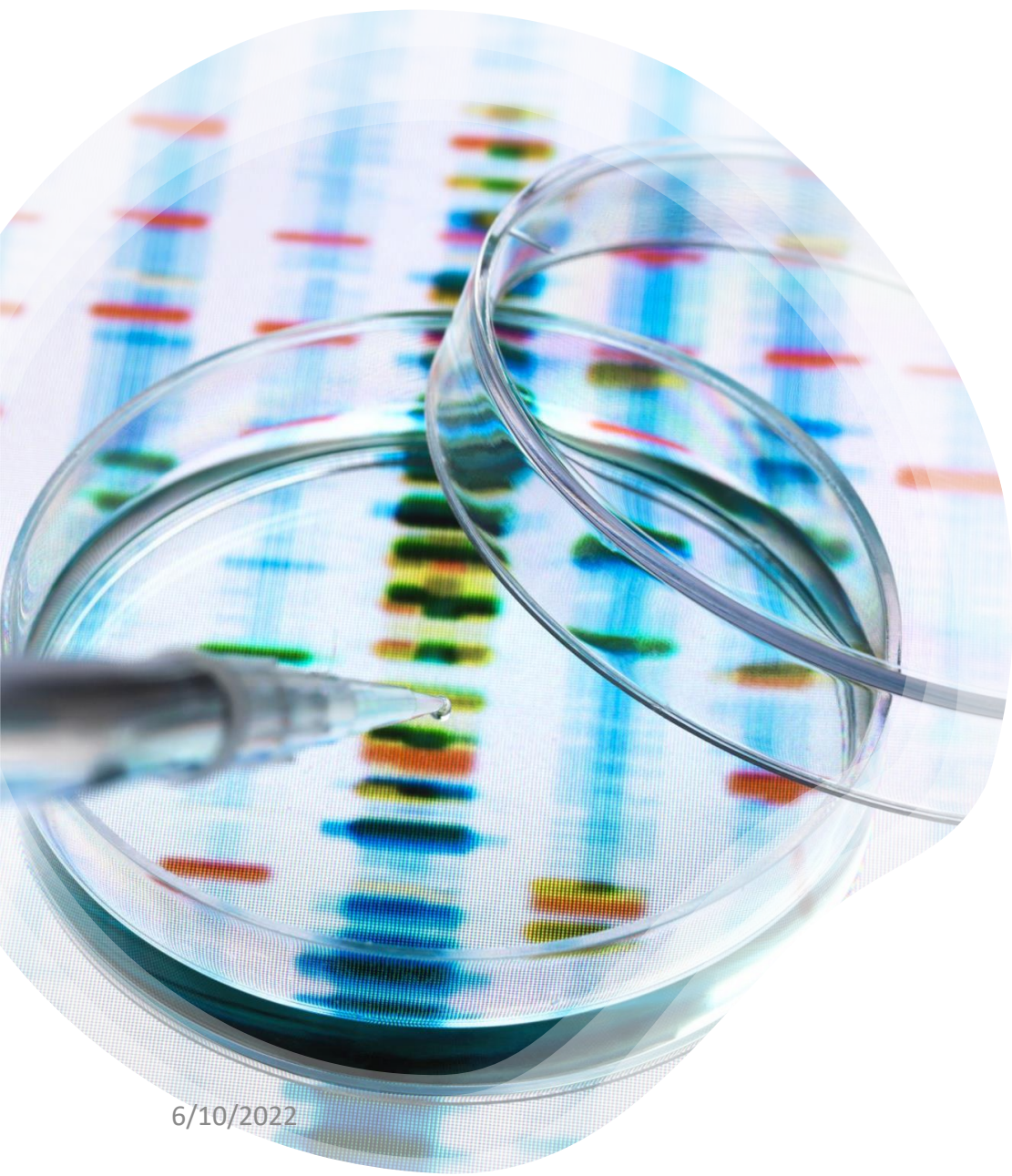
Matrix Multiplication Performance

- It takes <25 lines of code to write a Triton kernel on par with cuBLAS
- Arbitrary ops can be “fused” before/after the GEMM while the data is still on-chip, leading to large speedups over PyTorch
- More examples
 - [Tutorials – Triton documentation \(triton-lang.org\)](https://triton-lang.org)



Triton Future Work

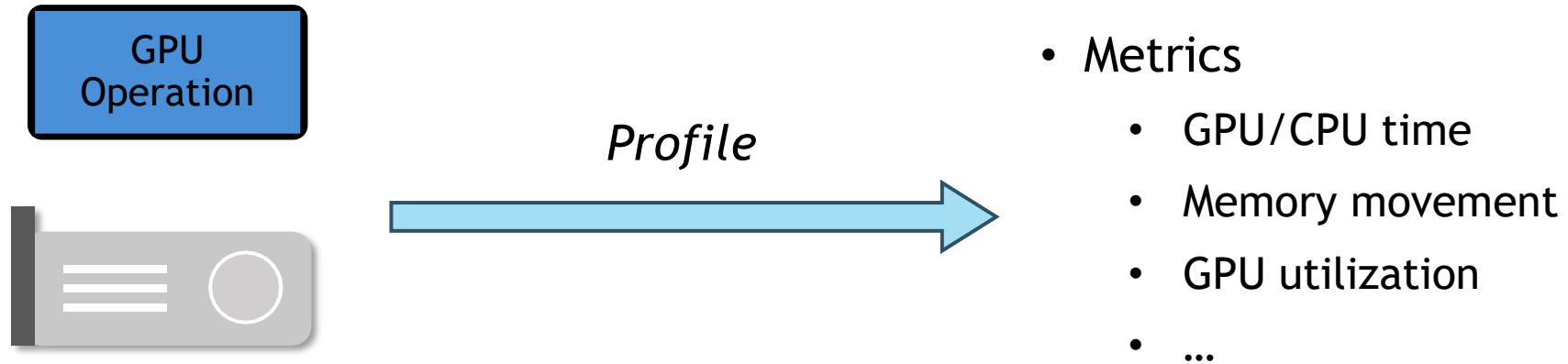
- Rewrite with MLIR
- Enhance debugging utility
- Support Hopper GPUs



GPU Performance Analysis

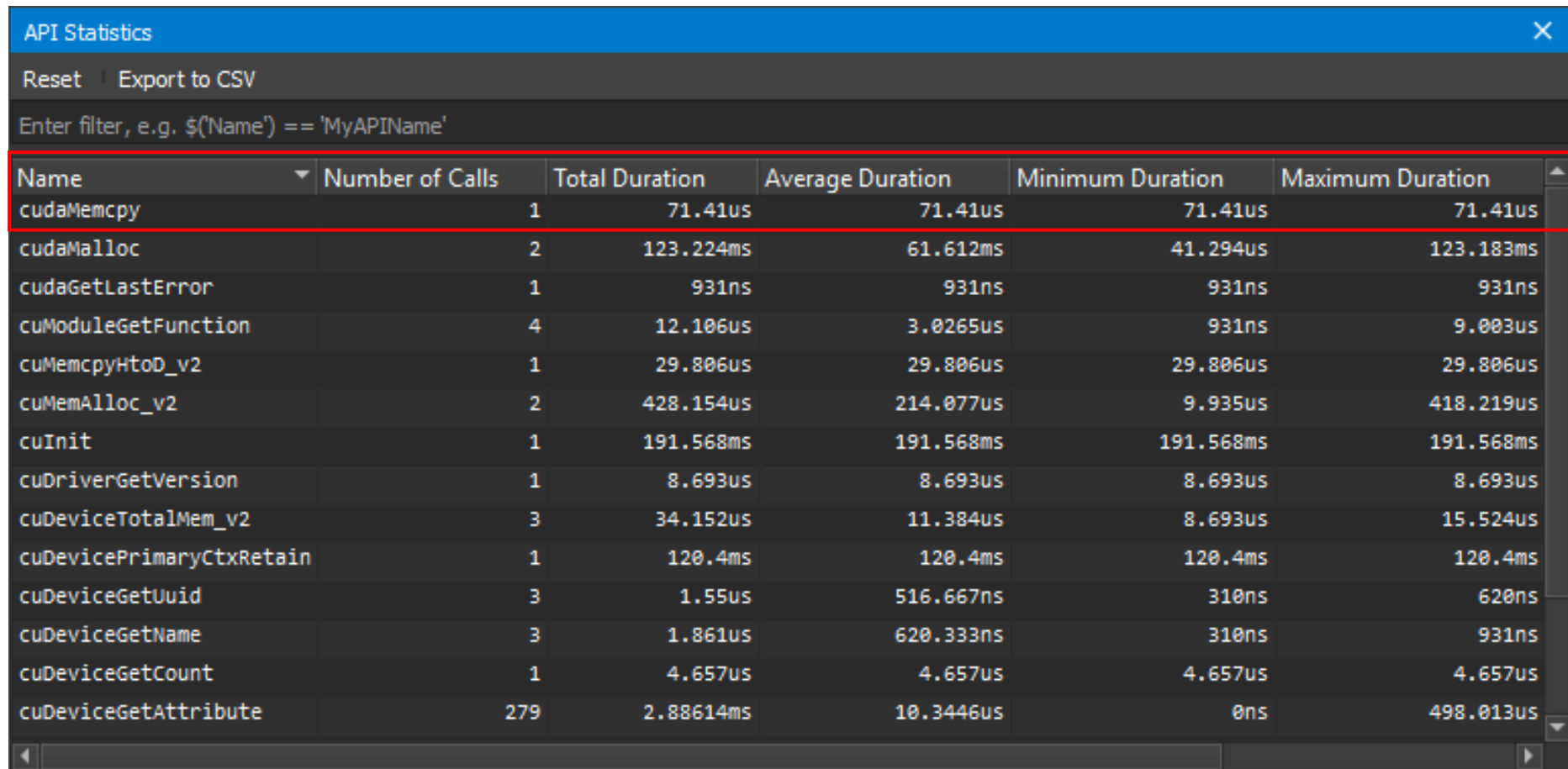
GPU Performance Tools

- Measurement Modalities
 - Interception of GPU operations
 - Instrumentation within GPU kernels
 - Instruction sampling in GPU kernels



- NVIDIA Nsight Systems/Compute
- AMD RocTracer/RocProfiler
- Intel VTune
- ...

Flat Profile View



The screenshot shows the 'API Statistics' window in Nsight Compute. It features a search bar with the placeholder text 'Enter filter, e.g. \$('Name') == 'MyAPIName'', and buttons for 'Reset' and 'Export to CSV'. Below is a table with columns: Name, Number of Calls, Total Duration, Average Duration, Minimum Duration, and Maximum Duration. The first row, 'cudaMemcpy', is highlighted with a red border. The table lists various CUDA API calls and their performance metrics.

Name	Number of Calls	Total Duration	Average Duration	Minimum Duration	Maximum Duration
cudaMemcpy	1	71.41us	71.41us	71.41us	71.41us
cudaMalloc	2	123.224ms	61.612ms	41.294us	123.183ms
cudaGetLastError	1	931ns	931ns	931ns	931ns
cuModuleGetFunction	4	12.106us	3.0265us	931ns	9.003us
cuMemcpyHtoD_v2	1	29.806us	29.806us	29.806us	29.806us
cuMemAlloc_v2	2	428.154us	214.077us	9.935us	418.219us
cuInit	1	191.568ms	191.568ms	191.568ms	191.568ms
cuDriverGetVersion	1	8.693us	8.693us	8.693us	8.693us
cuDeviceTotalMem_v2	3	34.152us	11.384us	8.693us	15.524us
cuDevicePrimaryCtxRetain	1	120.4ms	120.4ms	120.4ms	120.4ms
cuDeviceGetUuid	3	1.55us	516.667ns	310ns	620ns
cuDeviceGetName	3	1.861us	620.333ns	310ns	931ns
cuDeviceGetCount	1	4.657us	4.657us	4.657us	4.657us
cuDeviceGetAttribute	279	2.88614ms	10.3446us	0ns	498.013us

Nsight Compute Profiling

Profile View Using HPCToolkit

```

246 // Return the total cross section for this energy group
247 HOST_DEVICE
248 double NuclearData::getReactionCrossSection(
249     unsigned int reactIndex, unsigned int isotopeIndex, unsigned int group)
250 {
251     qs_assert(isotopeIndex < _isotopes.size());
252     qs_assert(reactIndex < _isotopes[isotopeIndex]._species[0]._reactions.size());
253     return _isotopes[isotopeIndex]._species[0]._reactions[reactIndex].getCrossSection(group);
254 }

```

Scope	GINs:Sum (l)	GINs:STL_ANY:Sum (l)
loop at main.cc: 159	1.07e+11 100 %	9.83e+10 100 %
loop at main.cc: 163	1.07e+11 100 %	9.83e+10 100 %
193: [l] CycleTrackingKernel(MonteCarlo*, int, ParticleVault*, ParticleVault*)	1.07e+11 100 %	9.83e+10 100 %
127: _device_stub_Z19CycleTrackingKernelP10MonteCarloIP13ParticleVaultS2_(MonteCarlo*, int, Parti	1.07e+11 100 %	9.83e+10 100 %
CPU Calling Context 14: [l] cudaLaunchKernel<char>	1.07e+11 100 %	9.83e+10 100 %
GPU API Node 209: <gpu kernel>	1.07e+11 100 %	9.83e+10 100 %
174: CycleTrackingKernel(MonteCarlo*, int, ParticleVault*, ParticleVault*)	1.07e+11 100 %	9.83e+10 100 %
132: CycleTrackingGuts(MonteCarlo*, int, ParticleVault*, ParticleVault*)	1.06e+11 100.0	9.82e+10 100.0
loop at CycleTracking.cc: 118	8.90e+10 83.5%	8.08e+10 82.2%
63: CollisionEvent(MonteCarlo*, MC_Particle&, unsigned int)	4.99e+10 46.9%	4.49e+10 45.7%
[l] inlined from QS_Vector.hh: 94	3.76e+10 35.3%	3.34e+10 34.0%
loop at QS_Vector.hh: 94	3.61e+10 33.9%	3.20e+10 32.5%
[l] inlined from CollisionEvent.cc: 71	3.58e+10 33.6%	3.17e+10 32.3%
loop at CollisionEvent.cc: 71	3.42e+10 32.1%	3.03e+10 30.9%
73: macroscopicCrossSection(MonteCarlo*, int, int, int, int, int)	3.11e+10 29.2%	2.78e+10 28.3%
[l] inlined from MacroscopicCrossSection.cc: 45	2.57e+10 24.1%	2.31e+10 23.5%
41: NuclearData::getReactionCrossSection(unsigned int, unsigned	1.69e+10 15.9%	1.56e+10 15.9%
[l] inlined from NuclearData.cc: 194	9.36e+09 8.8%	8.70e+09 8.9%
GPU Hotspot NuclearData.cc: 253	9.06e+09 8.5%	8.44e+09 8.6%

HPCToolkit for Deep Learning Applications

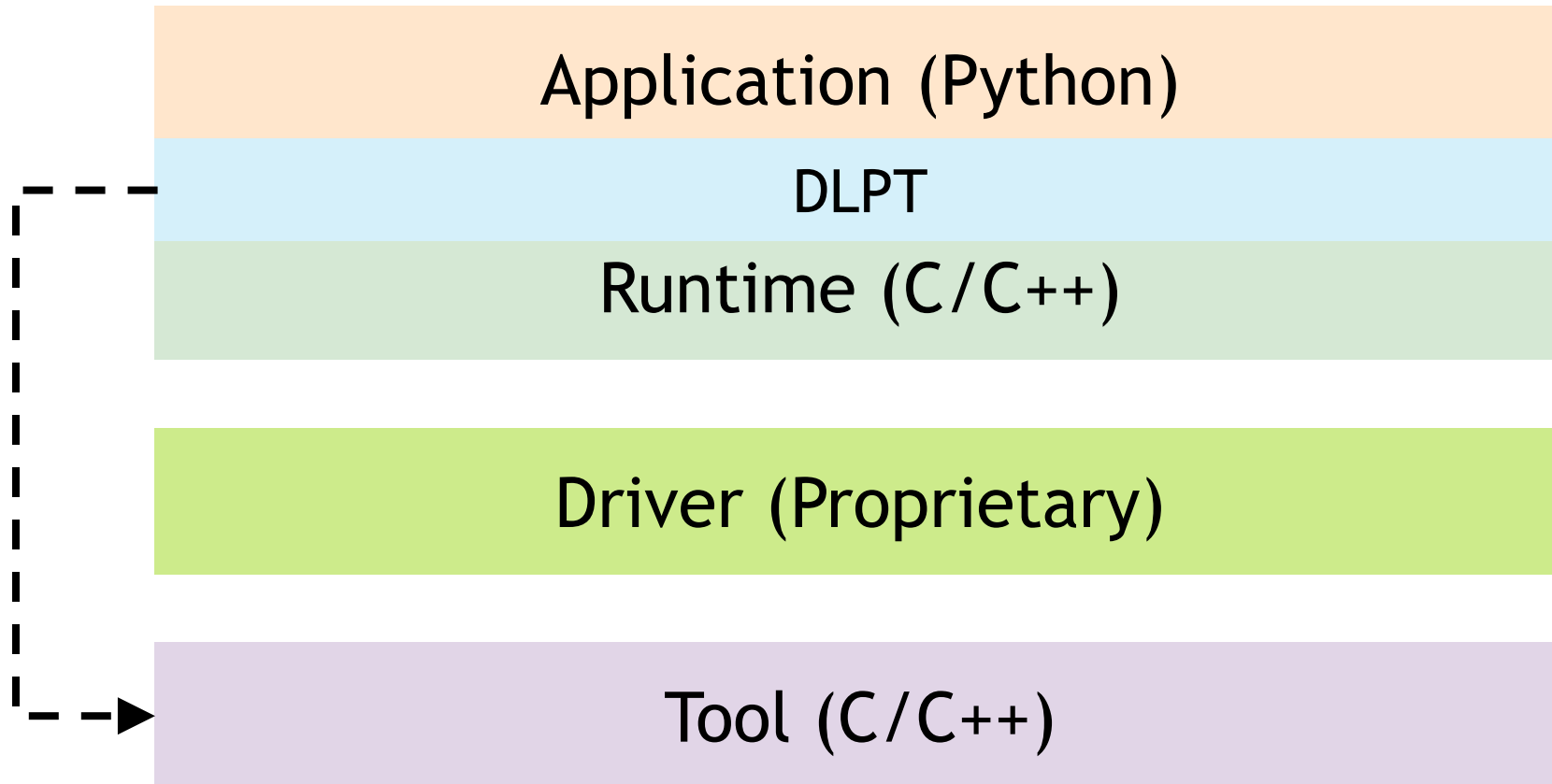
PyTorch-MNIST

Top-down view Bottom-up view Flat view		
Scope		
		GKER (sec):Sum (l)
Experiment Aggregate Metrics		1.27e-02 100.0%
<thread root>		7.62e-03 60.0%
<program root>		5.08e-03 40.0%
530: Py_BytesMain [python3.8]		5.08e-03 40.0%
1137: Py_RunMain.cold.2916 [python3.8]		5.08e-03 40.0%
[l] pymain_run_python		5.08e-03 40.0%
[l] pymain_run_file		5.08e-03 40.0%
[l] inlined from main.c: 347		5.08e-03 40.0%
387: PyRun_SimpleFileExFlags [python3.8]		5.08e-03 40.0%
428: PyRun_FileExFlags [python3.8]		5.08e-03 40.0%
1063: run_mod [python3.8]		5.08e-03 40.0%
1147: run_eval_code_obj [python3.8]		5.08e-03 40.0%
1125: PyEval_EvalCode [python3.8]		5.08e-03 40.0%
718: PyEval_EvalCodeEx [python3.8]		5.08e-03 40.0%
4327: _PyEval_EvalCodeWithName [python3.8]		5.08e-03 40.0%
4298: _sre_SRE_Match_expand [python3.8]		5.08e-03 40.0%
[l] inlined from ceval.c: 1239		5.08e-03 40.0%
loop at ceval.c: 1239		5.08e-03 40.0%
loop at ceval.c: 1323		5.08e-03 40.0%
[l] call_function		5.08e-03 40.0%
[l] _PyObject_Vectorcall		5.08e-03 40.0%
[l] inlined from abstract.h: 123		5.08e-03 40.0%
127: _PyFunction_Vectorcall.localalias.355 [python3.8]		5.08e-03 40.0%
[l] function_code_fastcall		5.08e-03 40.0%
[l] inlined from call.c: 279		5.08e-03 40.0%
283: _sre_SRE_Match_expand [python3.8]		5.08e-03 40.0%
[l] inlined from ceval.c: 1239		5.08e-03 40.0%

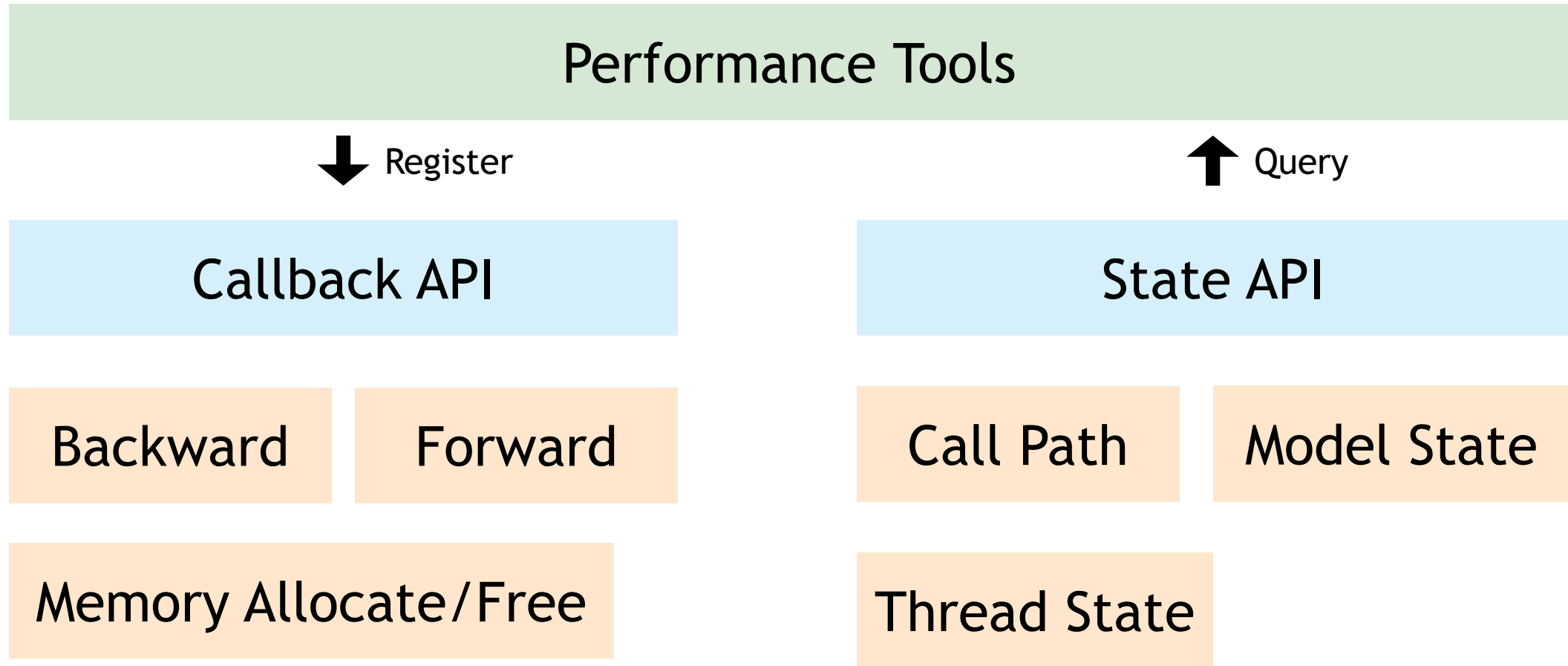
Low level Calling Context

No Application-level Information

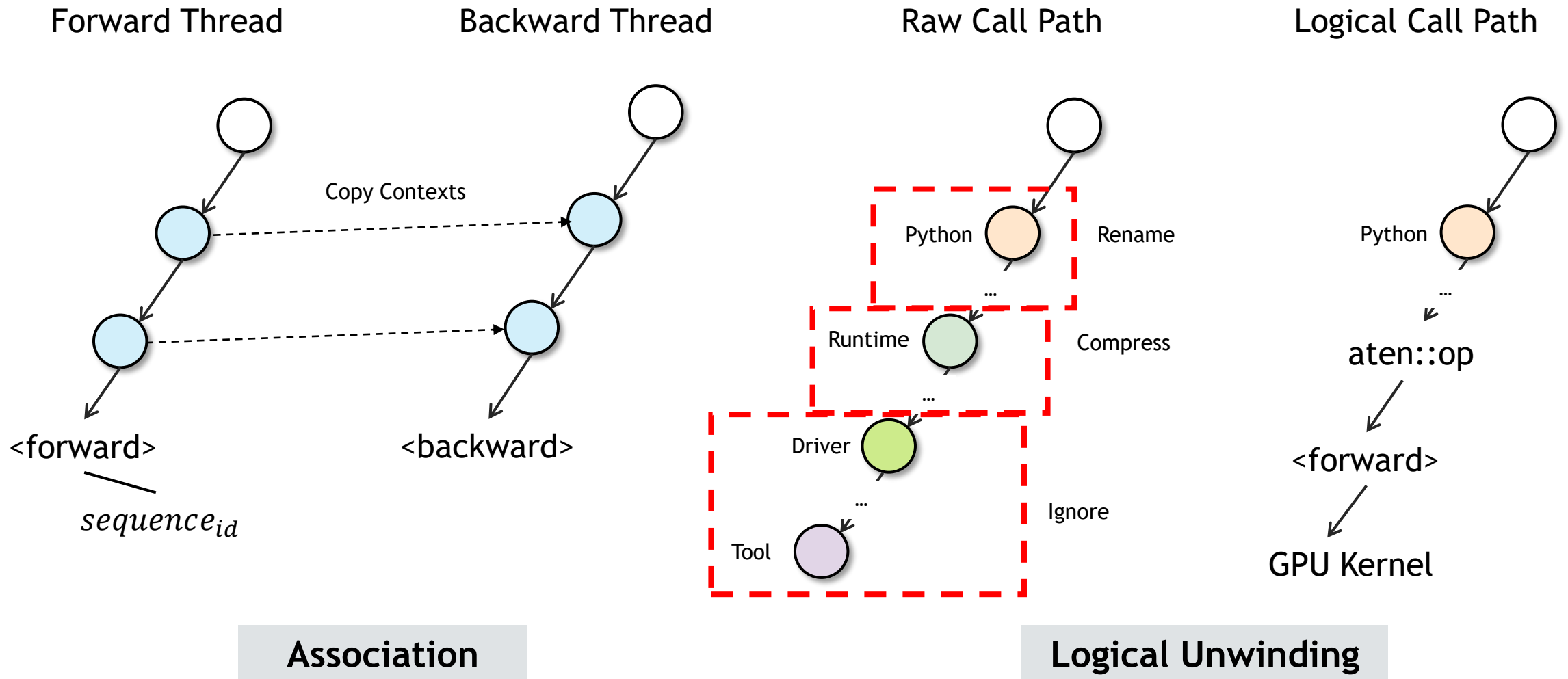
Deep Learning Profiling Interface (DLPT)



DLPT Components



Calling Context Manipulation



Profile View Using DLPT

mnist.py conv.py

```
438 def _conv_forward(self, input: Tensor, weight: Tensor, bias: Optional[Tensor]):
439     if self.padding_mode != 'zeros':
440         return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode=self.padding_mode),
441                         weight, bias, self.stride,
442                         _pair(0), self.dilation, self.groups)
443     return F.conv2d(input, weight, bias, self.stride,
444                     self.padding, self.dilation, self.groups)
```

Top-down view Bottom-up view Flat view

Scope

	GKER (sec):Sum (I)
Experiment Aggregate Metrics	1.27e-02 100.0%
<module>	1.20e-02 94.3%
137: main	1.20e-02 94.2%
128: train	1.20e-02 94.2%
42: _call_impl	1.16e-02 91.5%
1110: forward	1.16e-02 91.5%
24: _call_impl	7.37e-03 58.0%
1110: forward	7.37e-03 58.0%
447: _conv_forward	7.37e-03 58.0%
443: aten::conv2d	7.37e-03 58.0%
<backward>	4.89e-03 38.5%
<gpu kernel>	4.89e-03 38.5%
cutlass::Kernel<cutlass_80_tensorop_s1688gemm_64x64_16x6_nt_align4>(cutlass_80_tensor...	4.89e-03 38.5%

Python Calling Context for Forward Operation

Source Code Mapping

Phase Identification

Low-level GPU Kernels

ResNet - Nsight Systems

- GPU memory time

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
97.1	10,745,609	322	33,371.5	2,496.0	2,240	1,097,704	137,703.2	[CUDA memcpy HtoD]
2.8	309,316	54	5,728.1	5,568.5	4,865	6,656	408.1	[CUDA memcpy DtoD]
0.1	15,809	6	2,634.8	2,512.5	2,400	3,104	294.3	[CUDA memset]

- GPU kernel time

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
31.6	1,529,706	6	254,951.0	254,913.5	254,018	256,450	902.5	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_32x3_nn_align\$
10.5	507,041	53	9,566.8	9,281.0	7,392	19,680	1,869.7	void at::native::batch_norm_transform_input_kernel<float, float, float, \$
9.3	449,157	20	22,457.9	19,616.0	11,776	49,344	8,777.1	void at::native::im2col_kernel<float>(long, const T1 *, long, long, long\$
7.3	351,456	49	7,172.6	7,008.0	6,496	9,376	717.1	void at::native::vectorized_elementwise_kernel<(int)4, at::native::<unna\$
7.2	350,882	19	18,467.5	18,432.0	16,608	24,672	1,924.8	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_64x64_32x6_nn_align4>\$

ResNet - DLPT - Kernel Time

resnet.py

```
264 def forward_impl(self, x: Tensor) -> Tensor:
265     # See note [TorchScript super()]
266     x = self.conv1(x)
267     x = self.bn1(x)
268     x = self.relu(x)
269     x = self.maxpool(x)
270
271     x = self.layer1(x)
272     x = self.layer2(x)
273     x = self.layer3(x)
274     x = self.layer4(x)
275
276     x = self.avgpool(x)
277     x = torch.flatten(x, 1)
278     x = self.fc(x)
```

Source Code/Network Topology Mapping

Top-down view | Bottom-up view | Flat view

Scope

	GKER (sec):Sum (I)
283: _forward_impl	4.84e-03 100.0%
273: _call_impl	2.52e-03 52.0%
272: _call_impl	7.98e-04 16.5%
274: _call_impl	6.99e-04 14.5%
271: _call_impl	6.50e-04 13.4%

ResNet - DLPT - Memory Time

resnet.py ✕	
<pre>31# move the input and model to GPU for speed if available 32if torch.cuda.is_available() and device == 'cuda': 33 input_batch = input_batch.to(device) 34 model.to(device) 35</pre>	
Top-down view Bottom-up view Flat view	
Scope	GXCOPY (sec):Sum (I)
Experiment Aggregate Metrics	1.01e-02 100.0%
<module>	1.01e-02 100.0%
34: to	9.72e-03 96.5%
907: _apply	9.72e-03 96.5%
578: _apply	9.72e-03 96.5%
578: _apply	9.71e-03 96.3%
578: _apply	9.71e-03 96.3%
624: convert	8.90e-03 88.3%
905: aten::to	8.90e-03 88.3%
<forward>	8.90e-03 88.3%

DLPT Future Work

- Profile distributed systems
- Semantic analysis on calling context

More case studies on production deep learning applications are welcome!