

CUDA MODE: Lecture 1

Mark Saroufim

Logistics

Hosts: Andreas Köpf, Thomas Viehmann, Mark Saroufim

1 per 2 week on a CUDA topic: textbook chapter, pair programming session or project

Target audience is torch programmers tired of CUDA tutorial hell

Textbook: [Programming Massively Parallel Processors](#)

Additional resources here in [resource-stream](#)

All communication will happen on our [Discord](#)

Sessions will be recorded on <https://www.youtube.com/@CUDAMODE>

Goal of Lecture 1

1. Integrate a CUDA kernel inside a pytorch program
2. Learn how to profile it

Most of the code is here

<https://github.com/msaroufim/cudamc>
ture1

I believe thing I see



Start with something simple

```
=====
Profiling torch.square
=====
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:312] Completed Stage: Warm Up
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:318] Completed Stage: Collection
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:322] Completed Stage: Post Processing
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
aten::square	0.58%	20.000us	3.34%	115.000us	115.000us	17.000us	0.48%	3.506ms	3.506ms	1
aten::pow	2.09%	72.000us	2.76%	95.000us	95.000us	3.480ms	99.26%	3.489ms	3.489ms	1
aten::result_type	0.06%	2.000us	0.06%	2.000us	2.000us	5.000us	0.14%	5.000us	5.000us	1
aten::to	0.00%	0.000us	0.00%	0.000us	0.000us	4.000us	0.11%	4.000us	4.000us	1
cudaLaunchKernel	0.61%	21.000us	0.61%	21.000us	21.000us	0.000us	0.00%	0.000us	0.000us	1
cudaDeviceSynchronize	96.66%	3.332ms	96.66%	3.332ms	3.332ms	0.000us	0.00%	0.000us	0.000us	1

```
Self CPU time total: 3.447ms
Self CUDA time total: 3.506ms

=====
Profiling a * a
=====
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:312] Completed Stage: Warm Up
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:318] Completed Stage: Collection
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:322] Completed Stage: Post Processing
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
aten::mul	0.97%	33.000us	1.41%	48.000us	48.000us	3.455ms	100.00%	3.455ms	3.455ms	1
cudaLaunchKernel	0.44%	15.000us	0.44%	15.000us	15.000us	0.000us	0.00%	0.000us	0.000us	1
cudaDeviceSynchronize	98.59%	3.363ms	98.59%	3.363ms	3.363ms	0.000us	0.00%	0.000us	0.000us	1

```
Self CPU time total: 3.411ms
Self CUDA time total: 3.455ms

=====
Profiling a ** 2
=====
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:312] Completed Stage: Warm Up
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:318] Completed Stage: Collection
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:322] Completed Stage: Post Processing
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
aten::pow	1.37%	47.000us	1.69%	58.000us	58.000us	3.461ms	99.74%	3.470ms	3.470ms	1
aten::result_type	0.03%	1.000us	0.03%	1.000us	1.000us	5.000us	0.14%	5.000us	5.000us	1
aten::to	0.00%	0.000us	0.00%	0.000us	0.000us	4.000us	0.12%	4.000us	4.000us	1
cudaLaunchKernel	0.29%	10.000us	0.29%	10.000us	10.000us	0.000us	0.00%	0.000us	0.000us	1
cudaDeviceSynchronize	98.31%	3.378ms	98.31%	3.378ms	3.378ms	0.000us	0.00%	0.000us	0.000us	1

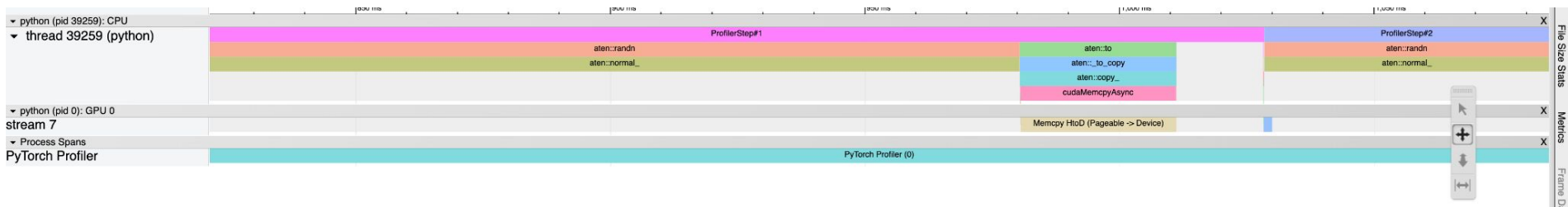
```
Self CPU time total: 3.436ms
Self CUDA time total: 3.470ms
```

```
9  def time_pytorch_function(func, input):
10     # CUDA IS ASYNC so can't use python time module
11     start = torch.cuda.Event(enable_timing=True)
12     end = torch.cuda.Event(enable_timing=True)
13
14     # Warmup
15     for _ in range(5):
16         func(input)
17
18     start.record()
19     func(input)
20     end.record()
21     torch.cuda.synchronize()
22     return start.elapsed_time(end)
23
24 b = torch.randn(10000, 10000).cuda()
25
26 def square_2(a):
27     return a * a
28
29 def square_3(a):
30     return a ** 2
31
32 time_pytorch_function(torch.square, b)
33 time_pytorch_function(square_2, b)
34 time_pytorch_function(square_3, b)
35
36 print("=====")
37 print("Profiling torch.square")
38 print("=====")
39
```

PyTorch profiler

Memcpy HtoD (Pageable -> Device)

- Host to device copy
- Pageable memory is on host but can be copied freely in out of RAM



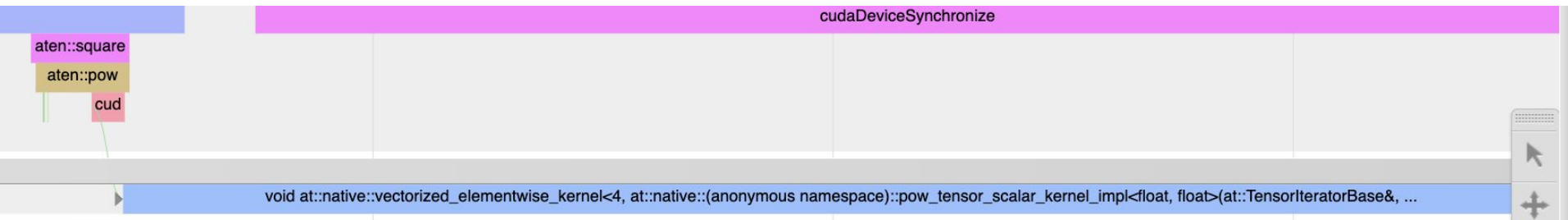
What can we learn?

Aten::square is a call to aten::pow

A cuda kernel gets launched called native_vectorized_elementwise_kernel<4, ..>

4 is the number of blocks

<https://github.com/pytorch/pytorch/blob/main/caffe2/utils/math/elementwise.cu>



Custom cpp extensions

```
hello_load_inline.py > ...  
1  import torch  
2  from torch.utils.cpp_extension import load_inline  
3  
4  cpp_source = """  
5      std::string hello_world() {  
6          return "Hello World!";  
7      }  
8      """  
9  
10 my_module = load_inline(  
11     name='my_module',  
12     cpp_sources=[cpp_source],  
13     functions=['hello_world'],  
14     verbose=True  
15 )  
16  
17 print(my_module.hello_world())
```

Codegen

```
#include <torch/extension.h>
```

```
std::string hello_world() {  
    return "Hello World!";  
}
```

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {  
    m.def("hello_world", torch::wrap_pybind_function(hello_world), "hello_world");  
}
```


How to run a CUDA kernel from pytorch

https://github.com/msaroufim/cudamodelecture1/blob/main/load_inline.py

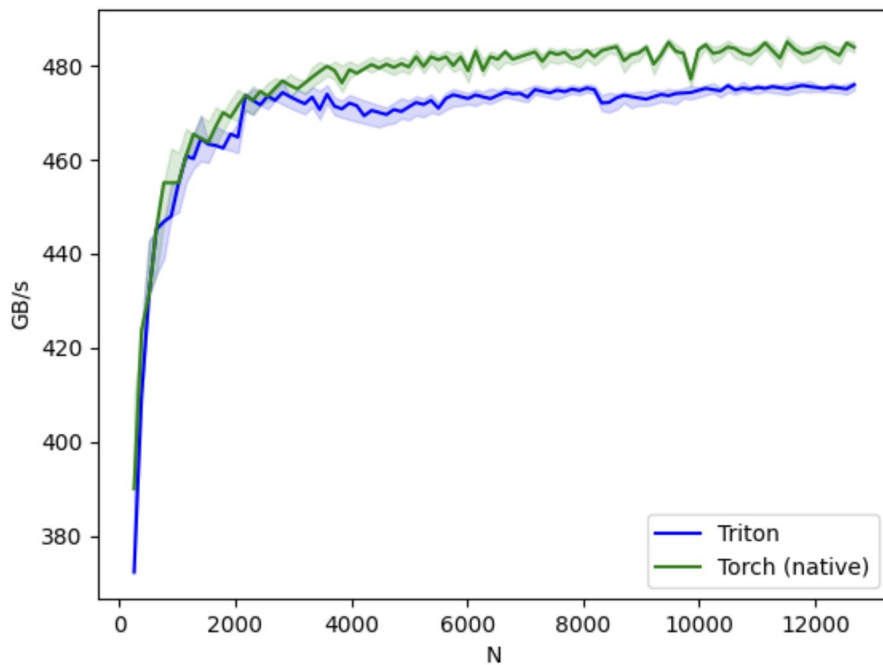
numba

<https://gist.github.com/msaroufim/6673c9e5c0c3d58740472601eac6d4df>

Integrate a triton kernel

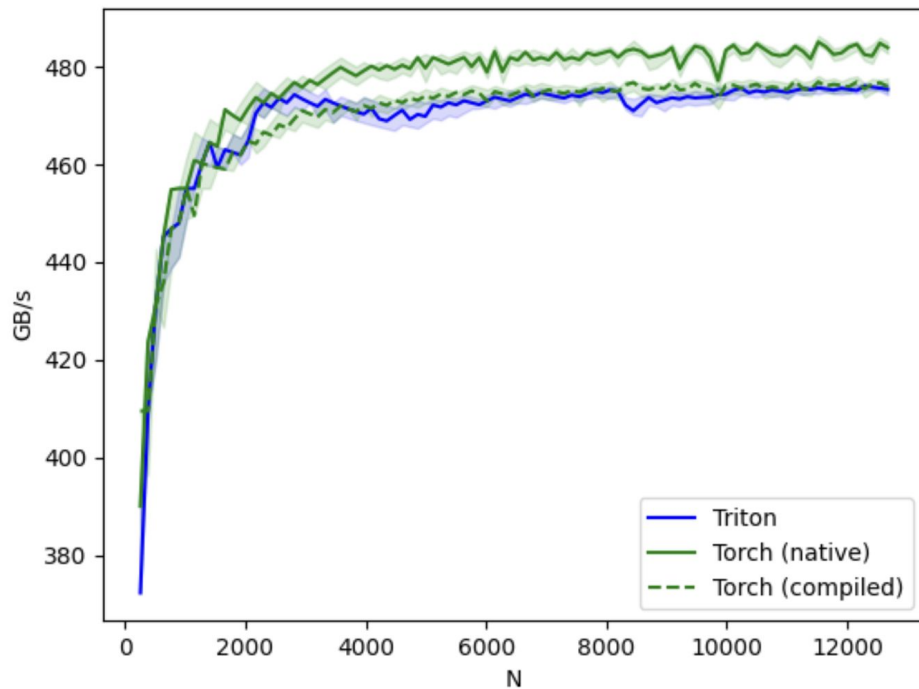
```
27 def square(x):
28     n_rows, n_cols = x.shape
29     # The block size is the smallest power of two greater than the number of columns in `x`
30     BLOCK_SIZE = triton.next_power_of_2(n_cols)
31     # Another trick we can use is to ask the compiler to use more threads per row by
32     # increasing the number of warps (`num_warps`) over which each row is distributed.
33     # You will see in the next tutorial how to auto-tune this value in a more natural
34     # way so you don't have to come up with manual heuristics yourself.
35     num_warps = 4
36     if BLOCK_SIZE >= 2048:
37         num_warps = 8
38     if BLOCK_SIZE >= 4096:
39         num_warps = 16
40     # Allocate output
41     y = torch.empty_like(x)
42     # Enqueue kernel. The 1D launch grid is simple: we have one kernel instance per row o
43     # f the input matrix
44     square_kernel[(n_rows, )](
45         y,
46         x,
47         x.stride(0),
48         y.stride(0),
49         n_cols,
50         num_warps=num_warps,
51         BLOCK_SIZE=BLOCK_SIZE,
52     )
53     return y
54
```

Triton kernel on A10G

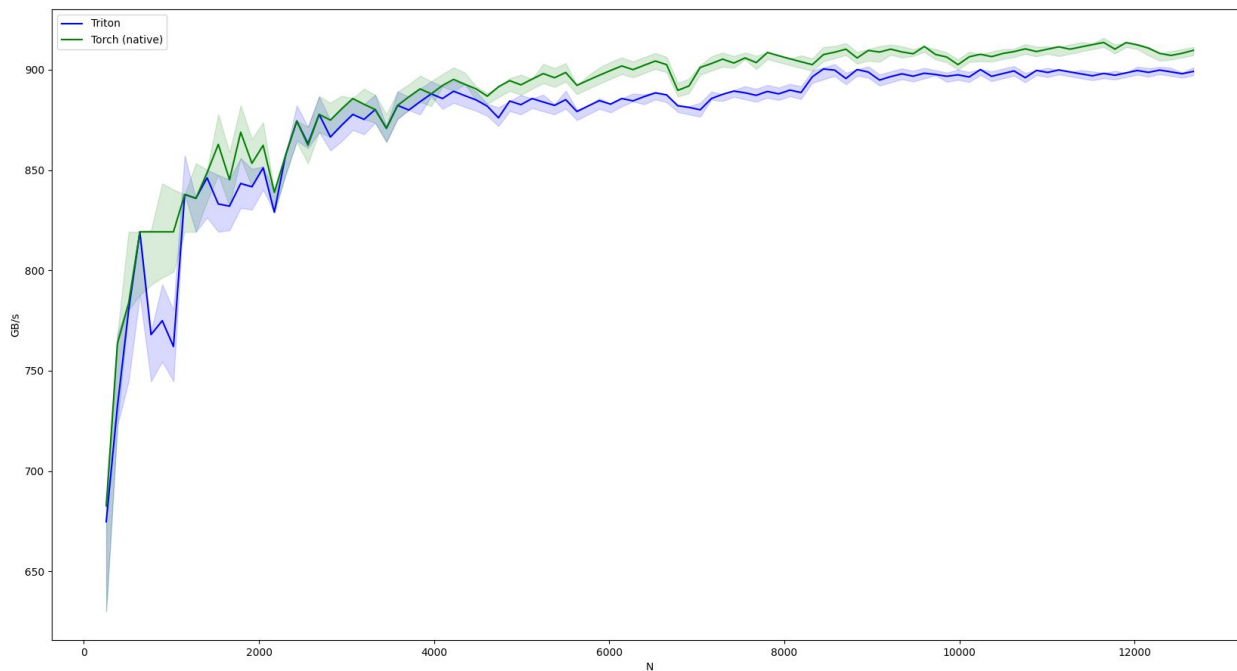


<https://gist.github.com/msaroufim/8649307ecdbb9309ced2d5106073bc0c>

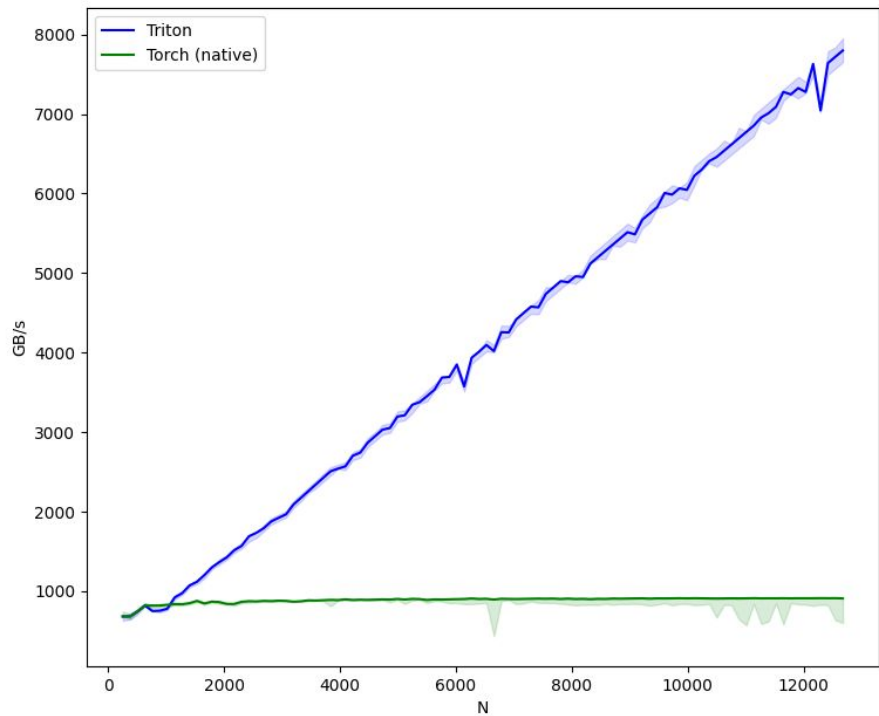
torch.compile



Results on 4090



After fixing the block size to 1024



Triton has a debugger now

```
triton.jit(interpret=True)
```

Almost everything is a WrappedTensor so inspect variables with var_name.tensor

<https://gist.github.com/msaroufim/f849df30687708782e0269c4b42264b1>

Look at PTX

https://github.com/msaroufim/cudamodelecture1/blob/main/square_kernel.ptx

8 registers with a self
multiplication for input

8 registers for output

```
.loc   1 19 26
```

```
mul.f32    %f9, %f1, %f1;
```

```
mul.f32    %f10, %f2, %f2;
```

```
mul.f32    %f11, %f3, %f3;
```

```
mul.f32    %f12, %f4, %f4;
```

```
mul.f32    %f13, %f5, %f5;
```

```
mul.f32    %f14, %f6, %f6;
```

```
mul.f32    %f15, %f7, %f7;
```

```
mul.f32    %f16, %f8, %f8;
```

Cheat: Generate a triton kernel

TORCH_LOGS="output_code" python compile_square.py

torch.compile(torch.square))

```
import triton
import triton.language as tl
from torch._inductor.ir import ReductionHint
from torch._inductor.ir import TileHint
from torch._inductor.triton_heuristics import AutotuneHint, pointwise
from torch._inductor.utils import instance_descriptor
from torch._inductor import triton_helpers

@pointwise(size_hints=[2097152], filename=__file__, meta={'signature': {0: '*fp32', 1: '*fp32', 2:
@triton.jit
def triton_(in_ptr0, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xnumel = 1423763
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[: ]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = tmp0 * tmp0
    tl.store(out_ptr0 + (x0), tmp1, xmask)
```

ncu profiler

```
ncu python train.py
```

```
ncu --set full -o output $(which python) train.py
```

https://github.com/msaroufim/cudamodelecture1/blob/main/ncu_logs

Contains actionable hints like

OPT This kernel grid is too small to fill the available resources on this device, resulting in only 0.4 full waves across all SMs. Look at Launch Statistics for more details.

FileConnectionDebugProfileToolsWindowHelp

ConnectDisconnectTerminateProfile Kernel

Some metric columns are hidden by default. Right-click any column header and use the column chooser to show them.

Project Explorer

Search project...Default Project

NVIDIA Nsight Compute

Jan 11 22:17

Page: SummaryResult: 1 - 576 - square_kernel_0d1d234

Add BaselineApply RulesOccupancy CalculatorSource Comparison

Copy as Image

Current

576 - square_kernel_0d1d234 (1823, 1, 1)x(128, 1, 1)

12.38 usecond25,830200 - NVIDIA GeForce RTX 4090

2.08 cycle/second8.9

[[10573] python3.10

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement (6583.68)	Compute Throughput	Memory Throughput	# Registers	Grid Size	Block Size
0	6.55	distribution_element...	void at::native::run...	5.98	0.39	51.98	43.49	38	768, 1, -	256, 1, -
1	58.08	square_kernel_0d1...	square_kernel_0d1...	12.38	6.19	7.08	89.96	20	1823, 1, -	128, 1, -

The following performance optimization opportunities were discovered for this result. Follow the rule links to see more context on the Details page.
Note: Speedup estimates provide upper bounds for the optimization potential of a kernel assuming its overall algorithmic structure is kept unchanged.

Tail Effect

Est. Speedup: 50.00%

A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full waves and a partial wave of 286 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 27.3%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid. See the [Tail Wave Model](#) description for more details on launch configurations.

Achieved Occupancy

Est. Speedup: 19.04%

The difference between calculated theoretical (100.0%) and measured achieved occupancy (72.7%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the [CUDA Best Practices Guide](#) for more details on optimizing occupancy.

Long Scoreboard Stalls

Est. Speedup: 19.04%

On average, each warp of this kernel spends 99.4 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. Find the instruction producing the data being waited upon to identify the culprit. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality (coalescing), or by changing the cache configuration. Consider moving frequently used data to shared memory. This stall type represents about 53.3% of the total average of 186.3 cycles between issuing two instructions.

Zoom in

Tail effect + Achieved occupancy 70%: Try padding (We can control)

Long scoreboard stalls: coalesce, use shared memory (Controlled by Triton :()

	CUDA	TRITON
Memory Coalescing	Manual	Automatic
Shared Memory Management	Manual	Automatic
Scheduling (Within SMs)	Manual	Automatic
Scheduling (Across SMs)	Manual	Manual

Compiler optimizations in CUDA vs Triton.

Activities

NVIDIA Nsight Compute

Jan 11 22:18

NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate Profile Kernel

Some metric columns are hidden by default. Right-click any column header and use the column chooser to show them.

Project Explorer

Welcome X: output.nvobj X: Untitled 1 X

Page: Details Result: 576 - square_kernel_0d1d234 Time: 12.38 usecond Cycles: 25,830 Reqs: 20 GPU: 0 - NVIDIA GeForce RTX 4090 SM Frequency: 2.08 cycle/usecond CC: 8.9 Process: [10573] python3.10

Copy as Image

Current

576 - square_kernel_0d1d234 (1823, 1, 1)x(128, 1, 1)

summary of the activity or the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (neoretical warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (active warps). Active warps that are not stalled (eligible warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (issued Warps). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]

Eligible Warps Per Scheduler [warp]

Issued Warp Per Scheduler

95.40

4.60

0.05

Issue Slot Utilization

Ext. Local Speedup: 19.04%

Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 21.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 3.56 active warps per scheduler, but only an average of 0.11 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, reduce the time the active warps are stalled by inspecting the top stall reasons on the > [Block State Reasons](#) and > [Source Comparison](#) sections.

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]

Warp Cycles Per Executed Instruction [cycle]

186.27 Avg. Active Threads Per Warp

194.84 Avg. Not Predicted Off Threads Per Warp

32

30.66

Long Scoreboard Stalls

Ext. Speedup: 19.04%

On average, each warp of this kernel spends 99.4 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. Find the instruction producing the data being waited upon to identify the culprit. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality (coalescing), or by changing the cache configuration. Consider moving frequently used data to shared memory. This stall type represents about 53.3% of the total average of 186.3 cycles between issuing two instructions.

Warp Stall

Check the > [Warp Stall Samples \(All Samples\)](#) table for the top stall locations in your source based on sampling data. The > [Kernel Profiling Data](#) provides more details on each stall reason.

Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]

Issued Instructions [inst]

488,564 Avg. Executed Instructions Per Scheduler [inst]

511,042 Avg. Issued Instructions Per Scheduler [inst]

954.23

998.13

FP32 Non-Fused Instructions

Ext. Speedup: 0.72%

This kernel executes 0 fused and 58336 non-fused FP32 instructions. By converting pairs of non-fused instructions to their > [fused](#), higher-throughput equivalent, the achieved FP32 performance could be increased by up to 50% (relative to its current performance). Check the Source page to identify where this kernel executes FP32 instructions.

NVLink Topology

NVLink Topology diagram shows logical NVLink connections with transmit/receive throughput.

NVLink Tables

Detailed tables with properties for each NVLink.

NUMA Affinity

Non-uniform memory access (NUMA) affinities based on compute and memory distances for all GPUs.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size

Registers Per Thread [register/thread]

Block Size

Threads [thread]

Waves Per SM

1,823 Function Cache Configuration

20 Static Shared Memory Per Block [byte/block]

128 Dynamic Shared Memory Per Block [byte/block]

233,344 Driver Shared Memory Per Block [kbyte/block]

1.19 Shared Memory Configuration Size [kbyte]

CachePref:None

0

0

1.02

32.77

Tail Effect

Ext. Speedup: 50.00%

A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full waves and a partial wave of 286 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 27.3%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid. See the > [Hardware Model](#) description for more details on launch configurations.

The following table lists the metrics that are key performance indicators:

Metric Name

Value

Guidance

launch_waves_per_multiprocessor 1.18685 Decrease the number of partial waves (the fractional part of the number of waves)

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]

Theoretical Active Warps per SM [warp]

100 Block Limit Registers [block]

48 Block Limit Shared Mem [block]

21

32

activities

NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate

Profile Kernel

Baselines

Metric Details

3

One metric column are hidden by default. Right-click any column header and use the column chooser to show them.

Project Explorer

Search project...
Default Project

Page: Source Result Time Cycles Regs GPU SM Frequency CC Process

Current576-square_kernel_0d1d234 (1823, 1, 1)(128, 1, 1)12.38 usecond25,830200-NVIDIA GeForce RTX 40902.08 cycle/insecond8.9[10573] python3.10

View: Source and SASS

Source: triton_square.pyFind...

Navigation: Instructions Executed

Source

triton_square.py

1 import triton

2 import triton.language as tl

3 import torch

4

5

6 @triton.jit()

7 def square_kernel(output_ptr, input_ptr, input_row_stride, output_row_stride, n_cols

8 # The rows of the softmax are independent, so we parallelize across those

9 row_idx = tl.program_id(0)

10 # The stride represents how much we need to increase the pointer to advance 1 r

11 row_start_ptr = input_ptr + row_idx * input_row_stride

12 # The block size is the next power of two greater than n_cols, so we can fit ea

13 # row in a single block

14 col_offsets = tl.arange(0, BLOCK_SIZE)

15 input_ptr = row_start_ptr + col_offsets

16 # Load the row into SRAM, using a mask since BLOCK_SIZE may be > than n_cols

17 row = tl.load(input_ptr, mask=col_offsets < n_cols, other=float('inf'))

18

19 square_output = row * row

20 # Write back output to DRAM

21 output_row_start_ptr = output_ptr + row_idx * output_row_stride

22 output_ptr = output_row_start_ptr + col_offsets

23 tl.store(output_ptr, square_output, mask=col_offsets < n_cols)

24

25 def square(x):

26 n_rows, n_cols = x.shape

27 # The block size is the smallest power of two greater than the number of column

28 BLOCK_SIZE = triton.next_power_of_2(n_cols)

29 # Another trick we can use is to ask the compiler to use more threads per row b

30 # Increasing the number of warps ('num_warps') over which each row is distribut

31 # You will see in the next tutorial how to auto-tune this value in a more natur

32 # way so you don't have to come up with manual heuristics yourself.

33 num_warps = 8

34 if BLOCK_SIZE >= 2048:

35 num_warps = 8

36 if BLOCK_SIZE >= 4096:

37 num_warps = 16

38 # Allocate output

39 y = torch.empty_like(x)

40 # Enqueue kernel. The ID launch grid is simple: we have one kernel instance per

41 # f the input matrix

Live arp Stall Sampling Registers (All Samples)

Instructions Executed

Address Space

Access Operation

Access Size

L2 Theoretical Sectors Global Excessive

1 0.91%

5 1.40%

8 3.17%

5.97%

10 1.90%

13.43%

10 1.40%

14 53.25%

44.78%

Global(7)

Load(8)

32(8)

50.80%

13 4.23%

11.94%

14 0.04%

4.48%

13 0.15%

1.49%

32 36.29%

13.43%

Global(7)

Store(7)

32(7)

50.80%

Source: square_kernel_0d1d234Find...

Navigation: Instructions Executed

Address Source

19 00007faf b5295520 LOP3.LUT R2, R7, #x200, R2,

20 00007faf b5295538 #P3 LDC.E R6, [R4, #4]

21 00007faf b5295540 LOP3.LUT R6, R7, #x300, R2,

22 00007faf b5295550 ISETP.GE.AND R3, PT, R0, c[

23 00007faf b5295560 ISETP.GE.AND R4, PT, R0, c[

24 00007faf b5295570 ISETP.GE.AND R5, PT, R0, c[

25 00007faf b5295580 LOP3.LUT R0, R7, #x300, R2,

26 00007faf b5295590 CS2R R0, SR2

27 00007faf b52955a0 CS2R R10, SR2

28 00007faf b52955b0 CS2R R12, SR2

29 00007faf b52955c0 ISETP.GE.AND R6, PT, R0, c[

30 00007faf b52955d0 IMAD.MOV.U32 R14, R2, R,

31 00007faf b52955e0 LDC.E R0, [R4, #4+0x200]

32 00007faf b52955f0 #P1 LDC.E R0, [R4, #4+0x000]

33 00007faf b5295600 #P2 LDC.E R10, [R4, #4+0x000]

34 00007faf b5295610 #P3 LDC.E R11, [R4, #4+0x000]

35 00007faf b5295620 #P4 LDC.E R12, [R4, #4+0x000]

36 00007faf b5295630 #P5 LDC.E R13, [R4, #4+0x000]

37 00007faf b5295640 #P6 LDC.E R14, [R4, #4+0x000]

38 00007faf b5295650 IMAD R2, R15, c[0x1][x17M]

39 00007faf b5295660 P/R R16, R0, R2, 0x1 P2R R2

40 00007faf b5295670 ISETP.GE.AND R0, PT, R7, c[

41 00007faf b5295680 IMAD.WIDE R2, R2, R7, c[0x

42 00007faf b5295690 IMAD.WIDE.U32 R2, R7, 0x0

43 00007faf b52956a0 SEL R0, R0, 0xffff00000, 1P

44 00007faf b52956b0 FMUL R7, R0, R0

45 00007faf b52956c0 #P0 STG.E [R2, #0], R7

46 00007faf b52956d0 ISETP.NE.AND R0, PT, R10, f

47 00007faf b52956e0 SEL R0, R0, 0xffff00000, 1P

48 00007faf b52956f0 SEL R0, R0, 0xffff00000, 1P

49 00007faf b5295700 SEL R10, R10, 0xffff00000, 1

50 00007faf b5295710 SEL R11, R11, 0xffff00000, 1

51 00007faf b5295720 SEL R12, R12, 0xffff00000, 1

52 00007faf b5295730 SEL R13, R13, 0xffff00000, 1

53 00007faf b5295740 FMUL R15, R0, R0

54 00007faf b5295750 FMUL R0, R0, R0

55 00007faf b5295760 FMUL R5, R10, R10

56 00007faf b5295770 FMUL R11, R11, R11

57 00007faf b5295780 FMUL R7, R12, R12

58 00007faf b5295790 FMUL R13, R13, R13

59 00007faf b52957a0 SEL R14, R14, 0xffff00000, 1

60 00007faf b52957b0 #P0 STG.E [R2, #4+0x200], R15

Live arp Stall Sampling Registers (All Samples)

Instructions Executed

Address Space

Access Operation

Access Size

L2 Theoretical Sectors Global Excessive

9 4.23%

10 0.04%

10 0.04%

8 0.04%

10 0.04%

10 0.04%

10 0.04%

10 0.04%

14 0.42%

14 0.68%

14 0.42%

14 0.30%

14 0.45%

14 0.26%

14 0.38%

14 0.45%

13 0.04%

13 0.04%

14 0.04%

14 0.15%

12 29.34%

13 0.04%

12 6.72%

11 5.40%

10 3.55%

10 6.99%

10 0.04%

10 0.00%

10 0.00%

11 0.04%

10 0.04%

11 2.10%

10 0.00%

10 2.11%

Inline Functions Source Markers

Select a source line in an active inline function to show additional information.