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/cudamode/tree/master/lecture1

I believe thing I see

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

Start with something simple

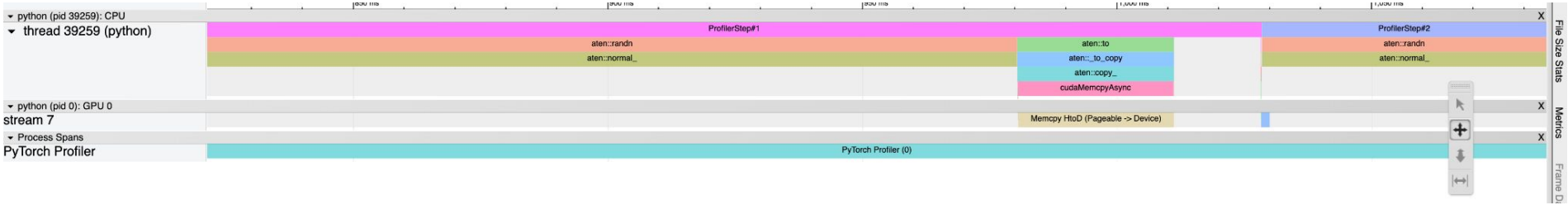
```
============
Profiling torch.square
============
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:312] Completed Stage: Warm Up
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:318] Completed Stage: Collection
STAGE:2024-01-12 20:00:20 2559:2559 ActivityProfilerController.cpp:322]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

```python
def time_pytorch_function(func, input):
    # CUDA IS ASYNC so can't use python time module
    start = torch.cuda.Event(enable_timing=True)
    end = torch.cuda.Event(enable_timing=True)

    # Warmup
    for _ in range(5):
        func(input)

    start.record()
    func(input)
    end.record()
    torch.cuda.synchronize()
    return start.elapsed_time(end)

b = torch.randn(10000, 10000).cuda()

def square_2(a):
    return a * a

def square_3(a):
    return a ** 2

time_pytorch_function(torch.square, b)
time_pytorch_function(square_2, b)
time_pytorch_function(square_3, b)

print("=============")
print("Profiling torch.square")
print("=============")
```

PyTorch profiler

Memcpy HtoD (Pageable -> Device)

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



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

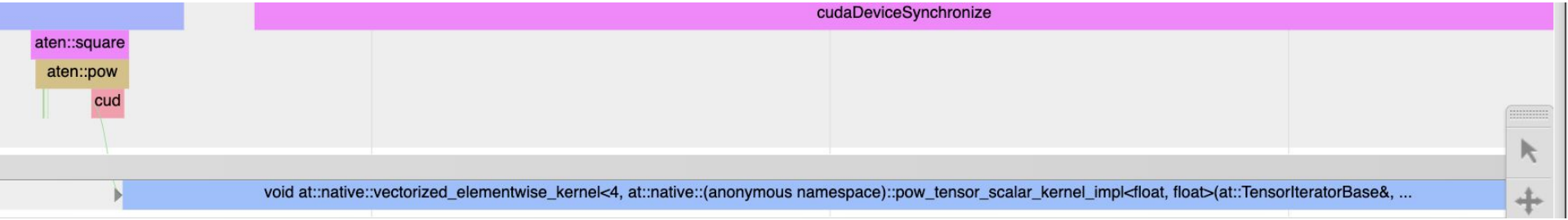
What can we learn?

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 > ...

```python
import torch
from torch.utils.cpp_extension import load_inline

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

my_module = load_inline(
    name='my_module',
    cpp_sources=[cpp_source],
    functions=['hello_world'],
    verbose=True
)

print(my_module.hello_world())
```

https://github.com/msaroufim/cudamodelecture1/tree/main/tmp

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

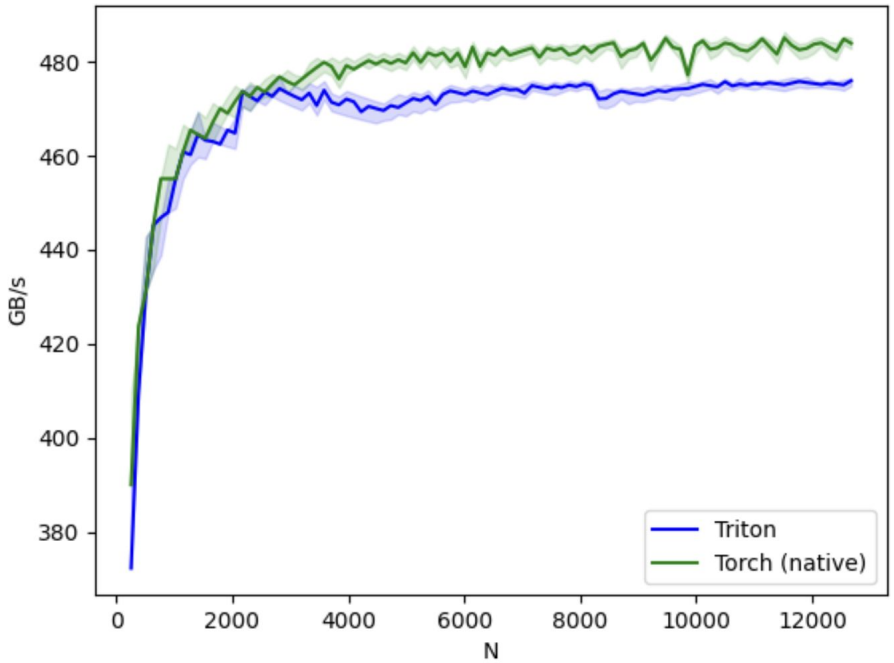
numba

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

Integrate a triton kernel

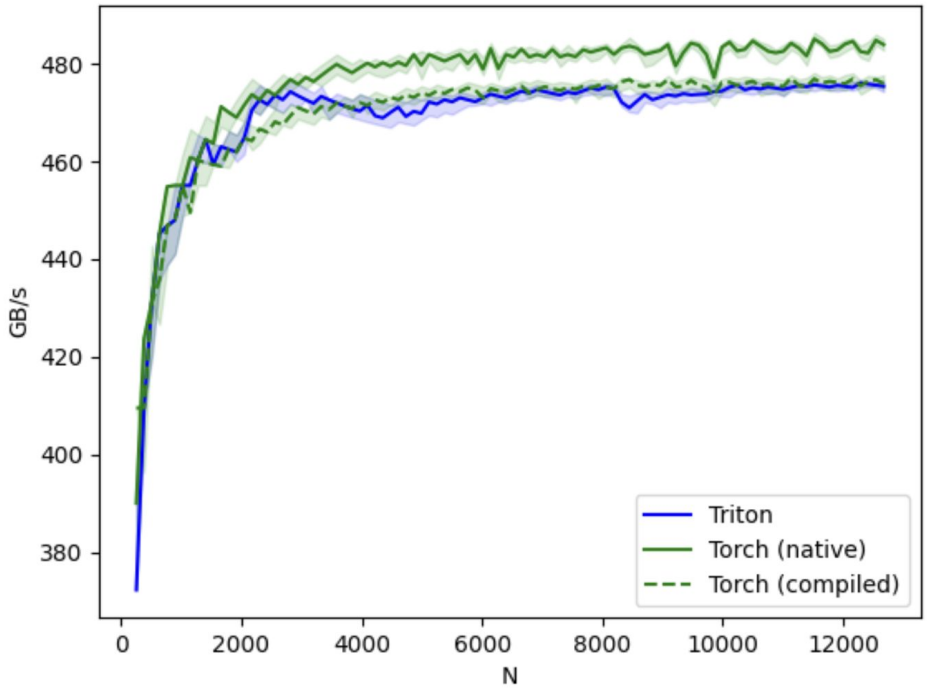
```
27  def square(x):
28      n_rows, n_cols = x.shape
29      # The block size is the smallest power of two greater than the number of columns in `x`
30      BLOCK_SIZE = triton.next_power_of_2(n_cols)
31      # Another trick we can use is to ask the compiler to use more threads per row by
32      # 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

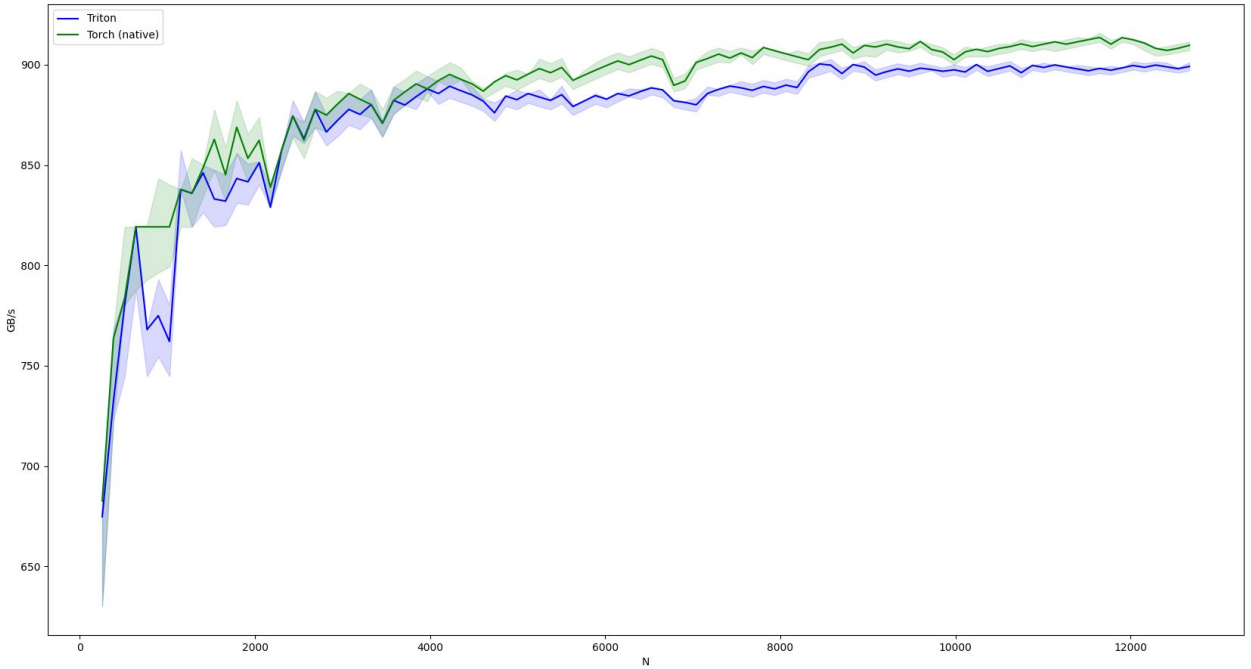


https://gist.github.com/msaroufim/8649307ecdbb9309ced2d5106073bc0c

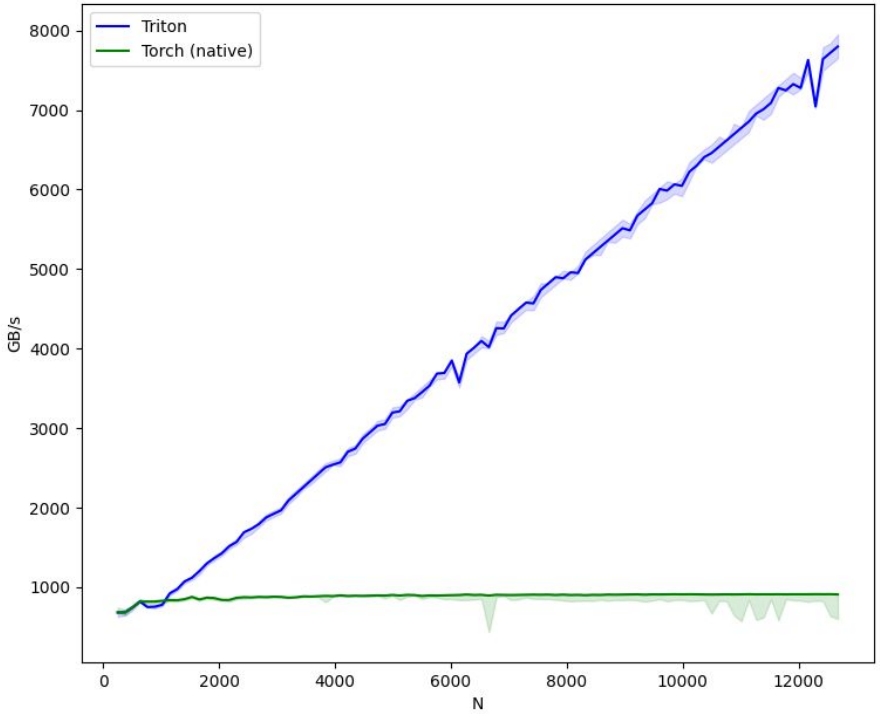
torch.compile



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

This means Triton is using 8 registers for storing inputs and another 8 registers for storing outputs.

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.

NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate Profile Kernel Baselines Metric Details

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

Welcome output.ncu-rep Untitled 1 *

Page: Summary Result: 1 - 576 - square_kernel_0d1d234 Add Baseline Apply Rules Occupancy Calculator Source Comparison Copy as image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	576 - square_kernel_0d1d234 (1823, 1, 1)x(128, 1, 1)	12.38 usecond	25,830	20	0 - NVIDIA GeForce RTX 4090	2.08 cycle/nsecond	8.9	[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_elemen...	void at::native::<un...	5.98	0.39	51.98	43.49	38	768, 1, ...	256, 1, ...
1	50.00	square_kernel_0d1...	square_kernel_0d1...	12.38	6.19	7.08	80.96	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 Hardware 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.

NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate Profile Kernel Baselines Metric Details

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

Welcome output.ncu-rep Untitled 1 *

Page: Details Result: 1 - 576 - square_kernel_0d1d234 Add Baseline Apply Rules Occupancy Calculator Source Comparison Copy as Image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	576 - square_kernel_0d1d234 (1823, 1, 1)x(128, 1, 1)	12.38 usecond	25,830	20	0 - NVIDIA GeForce RTX 4090	2.08 cycle/nsecond	8.9	[1573] python3.10

Summary of the activity of 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 (Theoretical 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 Warp). 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]	8.56	No Eligible [%]	95.40
Eligible Warps Per Scheduler [warp]	0.11	One or More Eligible [%]	4.60
Issued Warp Per Scheduler	0.05		

Issue Slot Utilization Est. 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 8.56 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 Warp State Statistics and Source Counters 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]	186.27	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	194.84	Avg. Not Predicated Off Threads Per Warp	30.66

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.

Warp Stall Check the Warp Stall Sampling (All Samples) table for the top stall locations in your source based on sampling data. The Kernel Profiling Guide 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]	488,564	Avg. Executed Instructions Per Scheduler [inst]	954.23
Issued Instructions [inst]	511,042	Avg. Issued Instructions Per Scheduler [inst]	998.13

FP32 Non-Fused Instructions Est. Speedup: 0.73%

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	1,823	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	20	Static Shared Memory Per Block [byte/block]	0
Block Size	128	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	233,344	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	1.19	Shared Memory Configuration Size [Kbyte]	32.77

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 Hardware Model description for more details on launch configurations.

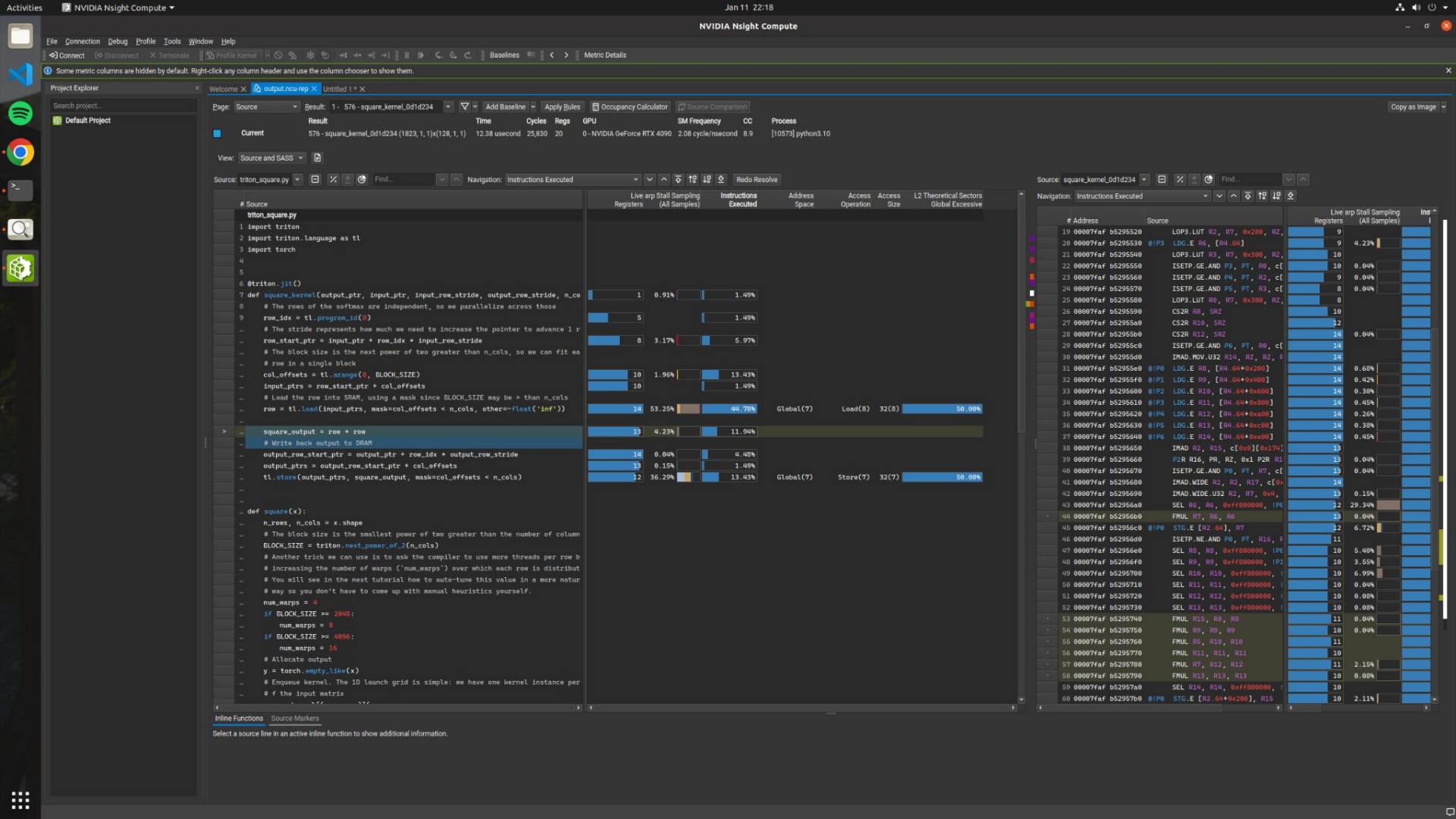
The following table lists the metrics that are key performance indicators:

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 [%]	100	Block Limit Registers [block]	21
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	32


Activities
NVIDIA Nsight Compute
Jan 11 22:18
NVIDIA Nsight Compute
File Connection Debug Profile Tools Window Help
Connect Disconnect Terminate Profile Kernel Baselines Metric Details
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
Welcome
output.ncu-rep
Untitled 1 *
Page: Source
Result: 1 - 576 - square_kernel_0d1d234
Add Baseline
Apply Rules
Occupancy Calculator
Source Comparison
Copy as image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 576 - square_kernel_0d1d234 (1823, 1, 1)x(128, 1, 1) 12.38 usecond 25,830 20 0 - NVIDIA GeForce RTX 4090 2.08 cycle/nsecond 8.9 [10573] python3.10
View: Source and SASS
Source: triton_square.py
Navigation: Instructions Executed
Redo Resolve
Source: square_kernel_0d1d234
Navigation: Instructions Executed
Source
triton_square.py
1 import triton
2 import triton.language as tl
3 import torch
6 @triton.jit()
7 def square_kernel(output_ptr, input_ptr, input_row_stride, output_row_stride, n_co
8 # The rows of the softmax are independent, so we parallelize across those
9 row_idx = tl.program_id(0)
# The stride represents how much we need to increase the pointer to advance 1 r
row_start_ptr = input_ptr + row_idx * input_row_stride
# The block size is the next power of two greater than n_cols, so we can fit ea
# row in a single block
col_offsets = tl.arange(0, BLOCK_SIZE)
input_ptrs = row_start_ptr + col_offsets
# Load the row into SRAM, using a mask since BLOCK_SIZE may be > than n_cols
row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-float('inf'))
square_output = row * row
# Write back output to DRAM
output_row_start_ptr = output_ptr + row_idx * output_row_stride
output_ptrs = output_row_start_ptr + col_offsets
tl.store(output_ptrs, square_output, mask=col_offsets < n_cols)
def square(x):
n_rows, n_cols = x.shape
# The block size is the smallest power of two greater than the number of column
BLOCK_SIZE = triton.next_power_of_2(n_cols)
# Another trick we can use is to ask the compiler to use more threads per row b
# increasing the number of warps (`num_warps`) over which each row is distribut
# You will see in the next tutorial how to auto-tune this value in a more natur
# way so you don't have to come up with manual heuristics yourself.
num_warps = 4
if BLOCK_SIZE >= 2048:
num_warps = 8
if BLOCK_SIZE >= 4096:
num_warps = 16
# Allocate output
y = torch.empty_like(x)
# Enqueue kernel. The 1D launch grid is simple: we have one kernel instance per
# f the input matrix
Live arp Stall Sampling (All Samples) Registers
Instructions Executed
Address Space
Access Operation
Access Size
L2 Theoretical Sectors Global Excessive
1 0.91% 1.49%
5 1.49%
8 3.17% 5.97%
10 1.96% 13.43%
10 1.49%
14 53.25% 44.78% Global(7) Load(8) 32(8) 50.00%
13 4.23% 11.94%
14 0.04% 4.48%
13 0.15% 1.49%
12 36.29% 13.43% Global(7) Store(7) 32(7) 50.00%
Address Source
19 00007faf b5295520 LOP3.LUT R2, R7, 0x280, RZ,
20 00007faf b5295530 @!P3 LDG.E R6, [R4.64] 9 4.23%
21 00007faf b5295540 LOP3.LUT R3, R7, 0x300, RZ, 10
22 00007faf b5295550 ISETP.GE.AND P3, PT, R0, c[10 0.04%
23 00007faf b5295560 ISETP.GE.AND P4, PT, R2, c[9 0.04%
24 00007faf b5295570 ISETP.GE.AND P5, PT, R3, c[8 0.04%
25 00007faf b5295580 LOP3.LUT R0, R7, 0x380, RZ, 8
26 00007faf b5295590 CS2R R8, SRZ 10
27 00007faf b52955a0 CS2R R10, SRZ 12
28 00007faf b52955b0 CS2R R12, SRZ 14 0.04%
29 00007faf b52955c0 ISETP.GE.AND P6, PT, R0, c[14
30 00007faf b52955d0 IMAD.MOV.U32 R14, RZ, RZ, F 14
31 00007faf b52955e0 @!P0 LDG.E R8, [R4.64+0x200] 14 0.68%
32 00007faf b52955f0 @!P1 LDG.E R9, [R4.64+0x400] 14 0.42%
33 00007faf b5295600 @!P2 LDG.E R10, [R4.64+0x600] 14 0.30%
34 00007faf b5295610 @!P3 LDG.E R11, [R4.64+0x800] 14 0.45%
35 00007faf b5295620 @!P4 LDG.E R12, [R4.64+0xa00] 14 0.26%
36 00007faf b5295630 @!P5 LDG.E R13, [R4.64+0xc00] 14 0.38%
37 00007faf b5295640 @!P6 LDG.E R14, [R4.64+0xe00] 14 0.45%
38 00007faf b5295650 IMAD R2, R15, c[0x0][0x174] 13
39 00007faf b5295660 P2R R16, PR, RZ, 0x1 P2R R1 13 0.04%
40 00007faf b5295670 ISETP.GE.AND P0, PT, R7, c[13 0.04%
41 00007faf b5295680 IMAD.WIDE R2, R2, R17, c[0> 14
42 00007faf b5295690 IMAD.WIDE.U32 R2, R7, 0x4, 13 0.15%
43 00007faf b52956a0 SEL R6, R6, 0xff800000, !P0 12 29.34%
44 00007faf b52956b0 FMUL R7, R6, R6 13 0.04%
45 00007faf b52956c0 @!P0 STG.E [R2.64], R7 12 6.72%
46 00007faf b52956d0 ISETP.NE.AND P0, PT, R16, R 11
47 00007faf b52956e0 SEL R8, R8, 0xff800000, !P0 10 5.40%
48 00007faf b52956f0 SEL R9, R9, 0xff800000, !P1 10 3.55%
49 00007faf b5295700 SEL R10, R10, 0xff800000, ! 10 6.99%
50 00007faf b5295710 SEL R11, R11, 0xff800000, ! 10 0.04%
51 00007faf b5295720 SEL R12, R12, 0xff800000, ! 10 0.08%
52 00007faf b5295730 SEL R13, R13, 0xff800000, ! 10 0.08%
53 00007faf b5295740 FMUL R15, R8, R8 11 0.04%
54 00007faf b5295750 FMUL R9, R9, R9 10 0.04%
55 00007faf b5295760 FMUL R5, R10, R10 11
56 00007faf b5295770 FMUL R11, R11, R11 10
57 00007faf b5295780 FMUL R7, R12, R12 11 2.15%
58 00007faf b5295790 FMUL R13, R13, R13 10 0.08%
59 00007faf b52957a0 SEL R14, R14, 0xff800000, ! 10
60 00007faf b52957b0 @!P0 STG.E [R2.64+0x200], R15 10 2.11%
Inline Functions
Source Markers
Select a source line in an active inline function to show additional information.