MSRA SH Triton Study Group
4. Nsight Compute

2025/08/08

What is NVIDIA Nsight Compute

- NVIDIA Nsight Compute is an interactive profiler for CUDA and NVIDIA OptiX that provides detailed performance metrics and API debugging.

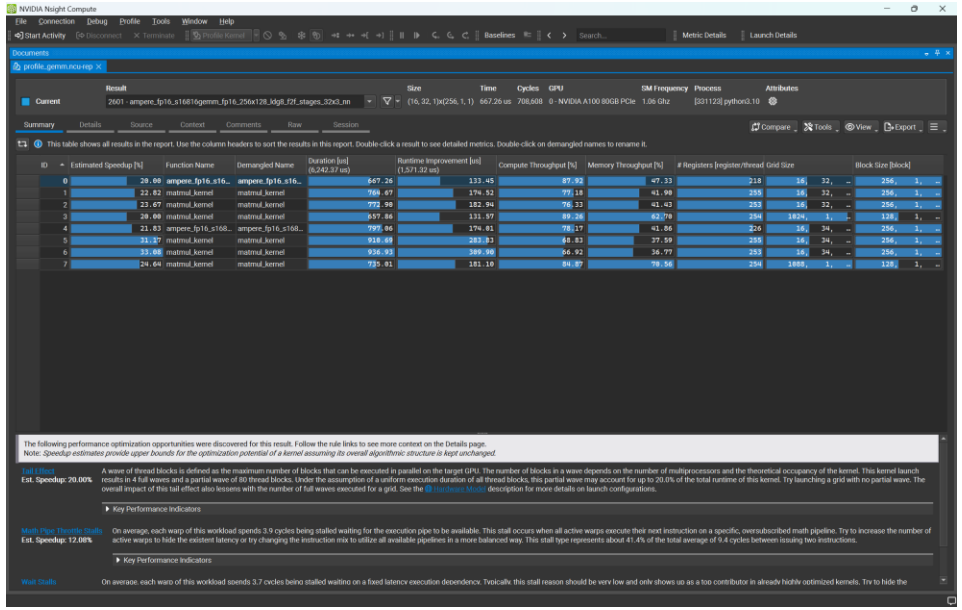
- Download: Getting Started with Nsight Compute | NVIDIA Developer
  - Host: on (Windows) Laptop
  - Target: on Linux (already installed on GCR A100 Sandbox: /usr/local/cuda/bin/ncu)

How to use NVIDIA Nsight Compute

- On target device: ncu + any program that uses GPUs

```
ncu="/usr/local/cuda/bin/ncu"
python="$(which python)"
sudo $ncu --set full -o profile_gemm -f $python ./call.py
```

- Download the log file (xxx.ncu-rep) and open with GUI



Example: Triton GeMM Kernel

- The triton GeMM kernel in the last class in not optimized

```
Problem size: (4096, 4096, 4096)
Torch GeMM Latency: 0.632 ms
Triton (Naive) GeMM Latency: 0.679 ms
```

Prepare for NCU profiling

- Get Autotune result

```
(moe) chengzhang@microsoft.com@GCRAZGDL1536:~/TritonStudyGroup/3_Triton_GeMM$
export TRITON_PRINT_AUTOTUNING=1
(moe) chengzhang@microsoft.com@GCRAZGDL1536:~/TritonStudyGroup/3_Triton_GeMM$
python test.py
Triton autotuning for function matmul_kernel finished after 1.67s; best config
 selected: BLOCK_SIZE_M: 128, BLOCK_SIZE_N: 256, BLOCK_SIZE_K: 64, num_warps:
8, num_ctas: 1, num_stages: 3, maxnreg: None;
```

- Rewrite triton kernel launch

```python
def triton_matmul(a: torch.Tensor, b: torch.Tensor) -> torch.Tensor:
    M, K = a.shape
    K2, N = b.shape
    assert K == K2, "Inner dimensions must match for matrix multiplication"
    c = torch.empty((M, N), dtype=a.dtype, device=a.device)
    BLOCK_SIZE_M = 128
    BLOCK_SIZE_N = 256
    BLOCK_SIZE_K = 64
    grid = (triton.cdiv(N, BLOCK_SIZE_N), triton.cdiv(M, BLOCK_SIZE_M), )
    matmul_kernel[grid](
        a, b, c, M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1),
        num_stages=3, num_warps=8,
        BLOCK_SIZE_M=BLOCK_SIZE_M, BLOCK_SIZE_N=BLOCK_SIZE_N, BLOCK_SIZE_K=BLOCK_SIZE_K,
    )
    return c
```

Run NCU profile and open replay

```
==PROF== Connected to process 21919 (/home/chengzhang/anaconda3/envs/moe/bin/python3.10)
==PROF== Profiling "ampere_fp16_s16816gemm_fp16_2..." - 0: 0%....50%....100% - 49 passes
==PROF== Profiling "matmul_kernel" - 1: 0%....50%....100% - 50 passes
==PROF== Profiling "matmul_kernel" - 2: 0%....50%....100% - 49 passes
==PROF== Profiling "matmul_kernel" - 3: 0%....50%....100% - 50 passes
==PROF== Profiling "ampere_fp16_s16816gemm_fp16_2..." - 4: 0%....50%....100% - 50 passes
==PROF== Profiling "matmul_kernel" - 5: 0%....50%....100% - 49 passes
==PROF== Profiling "matmul_kernel" - 6: 0%....50%....100% - 49 passes
==PROF== Profiling "matmul_kernel" - 7: 0%....50%....100% - 49 passes
==PROF== Disconnected from process 21919
```

NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Start Activity | Disconnect | Terminate | Profile Kernel | Baselines | Search... | Metric Details | Launch Details

Documents

profile_gemm.ncu-rep

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	2636 - matmul_kernel	(16, 32, 1)x(256, 1, 1)	764.67 us	811,401	0 - NVIDIA A100 80GB PCIe	1.05 Ghz	[331123] python3.10	
Baseline 1	2601 - ampere_fp16_s16816gemm_fp16_256x128_ldg8_f2f_stages_32x3_nn	(16, 32, 1)x(256, 1, 1)	667.26 us	708,608	0 - NVIDIA A100 80GB PCIe	1.06 Ghz	[331123] python3.10	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

GPU Speed Of Light Throughput

GPU Throughput Chart

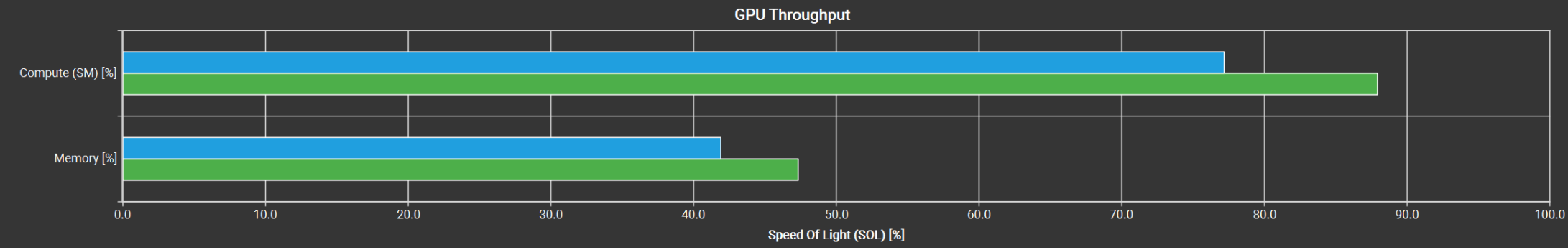
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	77.18 (-12.22%)	Duration [us]	764.67 (+14.60%)
Memory Throughput [%]	41.90 (-11.46%)	Elapsed Cycles [cycle]	811,401 (+14.51%)
L1/TEX Cache Throughput [%]	44.46 (-11.46%)	SM Active Cycles [cycle]	758,824.90 (+13.92%)
L2 Cache Throughput [%]	40.47 (-12.68%)	SM Frequency [Ghz]	1.05 (-0.59%)
DRAM Throughput [%]	15.90 (+3.53%)	DRAM Frequency [Ghz]	1.51 (-0.00%)

High Compute Throughput Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

Key Performance Indicators

Roofline Analysis The ratio of peak float (FP32) to double (FP64) performance on this device is 2:1. The workload achieved 0% of this device's FP32 peak performance and 0% of its FP64 peak performance. See the Profiling Guide for more details on roofline analysis.



NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Start Activity | Disconnect | Terminate | Profile Kernel | Baselines | Search... | Metric Details | Launch Details

Documents

profile_gemm.ncu-rep

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	2636 - matmul_kernel	(16, 32, 1)x(256, 1, 1)	764.67 us	811,401	0 - NVIDIA A100 80GB PCIe	1.05 Ghz	[331123] python3.10	
Baseline 1	2601 - ampere_fp16_s16816gemm_fp16_256x128_ldg8_f2f_stages_32x3_nn	(16, 32, 1)x(256, 1, 1)	667.26 us	708,608	0 - NVIDIA A100 80GB PCIe	1.06 Ghz	[331123] python3.10	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

Maximum Sampling Interval [cycle]	20,000 (+0.00%)	# Pass Groups	4 (+0.00%)
Maximum Buffer Size [Mbyte]	1.57 (+0.00%)	Dropped Samples [sample]	0 (+0.00%)

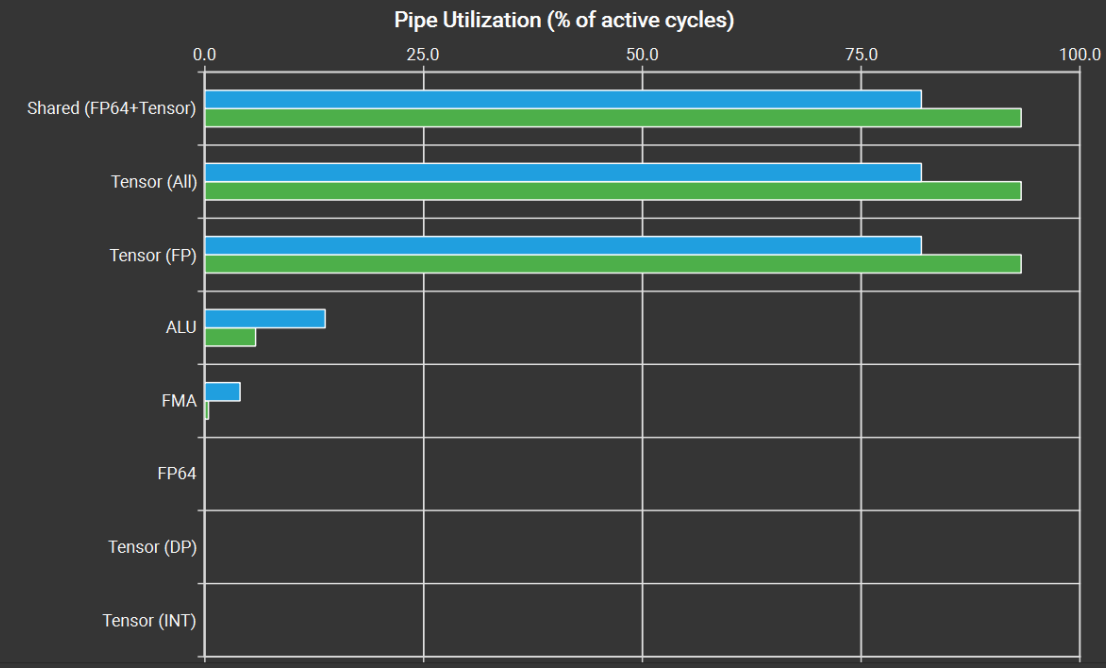
Compute Workload Analysis

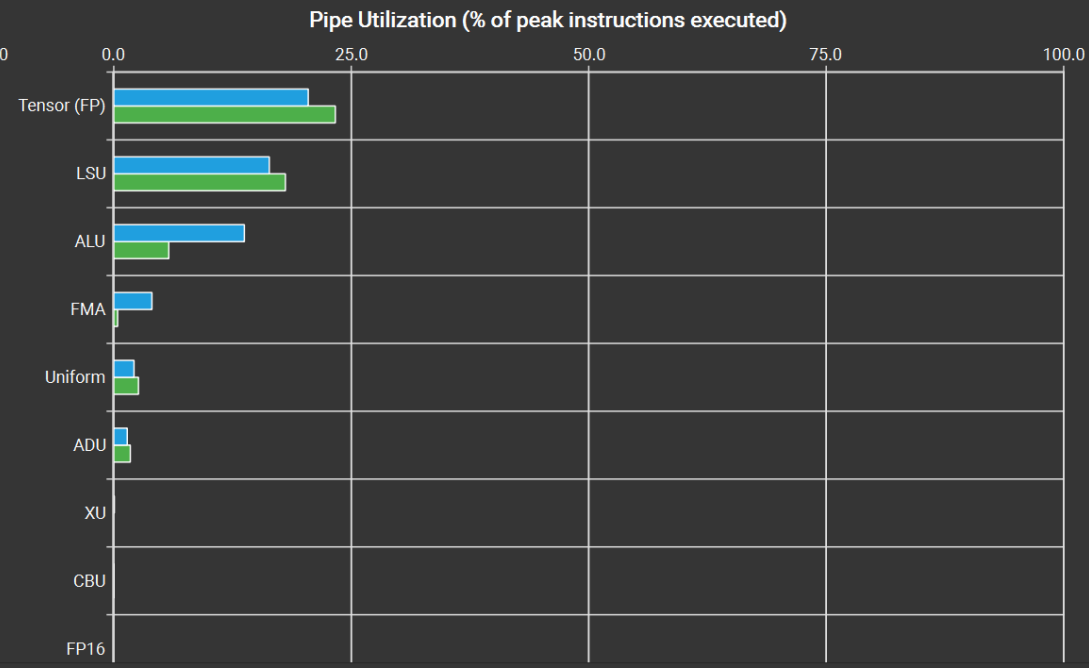
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	0.93 (+15.90%)	SM Busy [%]	81.89 (-12.22%)
Executed Ipc Active [inst/cycle]	0.99 (+15.90%)	Issue Slots Busy [%]	24.74 (+15.90%)
Issued Ipc Active [inst/cycle]	0.99 (+15.90%)		

Very High Utilization Shared is the highest-utilized pipeline (81.9%) based on active cycles, taking into account the rates of its different instructions. It is the logical sum of several other pipelines which can't achieve full utilization on their own. It executes 64-bit floating point and tensor operations. It's dominated by its Tensor (FP) sub-pipeline. The pipeline is over-utilized and likely a performance bottleneck. Based on the number of executed instructions, the highest utilized pipeline (20.5%) is Tensor (FP). It executes 16-bit floating point tensor operations. Comparing the two, the overall pipeline utilization appears to be caused by high-latency instructions. See the Profiling Guide or hover over the pipeline name to understand the workloads handled by each pipeline. The Instruction Statistics section shows the mix of executed instructions for this workload. Check the Warp State Statistics section for which reasons cause warps to stall.

Key Performance Indicators





NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Start Activity | Disconnect | Terminate | Profile Kernel | Baselines | Search... | Metric Details | Launch Details

Documents

profile_gemm.ncu-rep

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	2636 - matmul_kernel	(16, 32, 1)x(256, 1, 1)	764.67 us	811,401	0 - NVIDIA A100 80GB PCIe	1.05 Ghz	[331123] python3.10	
Baseline 1	2601 - ampere_fp16_s16816gemm_fp16_256x128_ldg8_f2f_stages_32x3_nn	(16, 32, 1)x(256, 1, 1)	667.26 us	708,608	0 - NVIDIA A100 80GB PCIe	1.06 Ghz	[331123] python3.10	

Summary | Details | Source | Context | Comments | Raw | Session

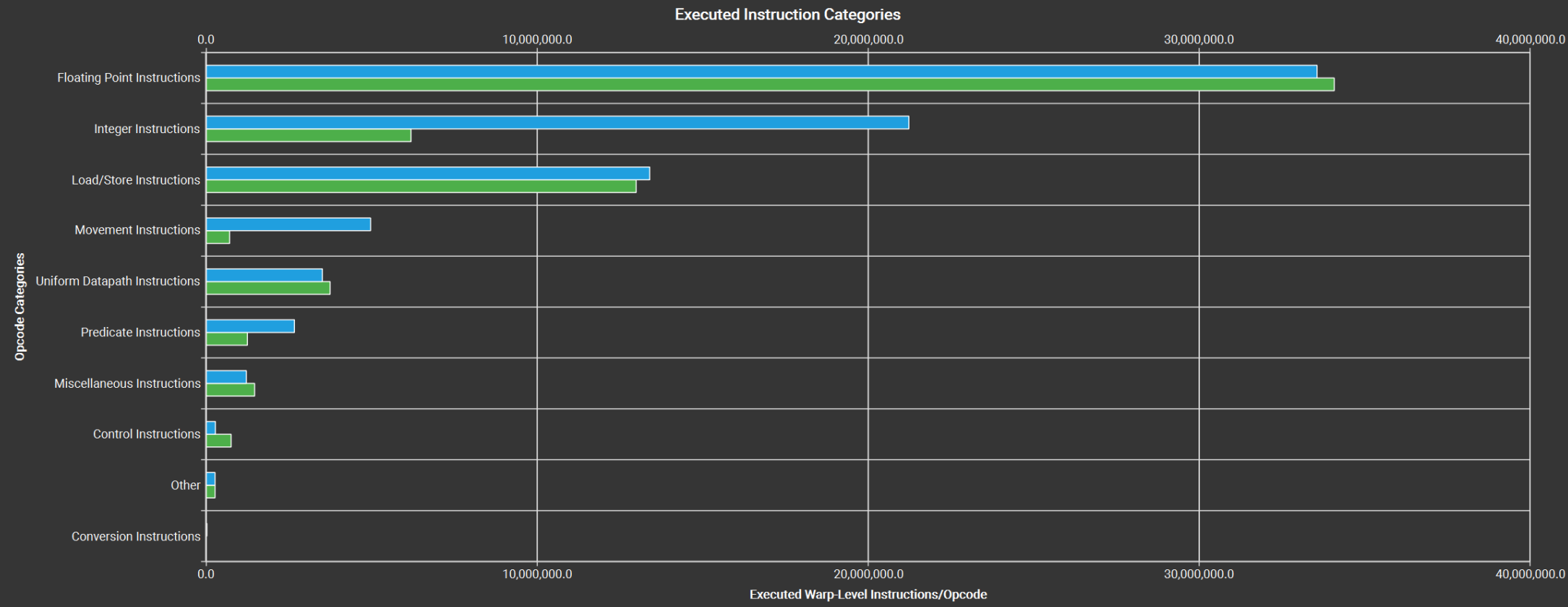
Compare | Tools | View | Export

Instruction Statistics

Opcode Category Chart

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]	81,084,416 (+32.03%)	Avg. Executed Instructions Per Scheduler [inst]	187,695.41 (+32.03%)
Issued Instructions [inst]	81,099,968 (+32.04%)	Avg. Issued Instructions Per Scheduler [inst]	187,731.41 (+32.04%)



NVLink Topology

Reduce ALU & Interger Instructions

```
# Create pointers for the first blocks of A and B
offs_m = (start_m + tl.arange(0, BLOCK_SIZE_M)) % M
offs_n = (start_n + tl.arange(0, BLOCK_SIZE_N)) % N
offs_k = tl.arange(0, BLOCK_SIZE_K)
a_ptrs = a_ptr + offs_m[:, None] * stride_am + offs_k[None, :] * stride_ak
b_ptrs = b_ptr + offs_k[:, None] * stride_bk + offs_n[None, :] * stride_bn
```

```
# Create pointers for the first blocks of A and B
offs_m = start_m + tl.arange(0, BLOCK_SIZE_M)
offs_n = start_n + tl.arange(0, BLOCK_SIZE_N)
offs_k = tl.arange(0, BLOCK_SIZE_K)
offs_m = tl.max_contiguous(tl.multiple_of(offs_m % M, BLOCK_SIZE_M), BLOCK_SIZE_M)
offs_n = tl.max_contiguous(tl.multiple_of(offs_n % N, BLOCK_SIZE_N), BLOCK_SIZE_N)
a_ptrs = a_ptr + offs_m[:, None] * stride_am + offs_k[None, :] * stride_ak
b_ptrs = b_ptr + offs_k[:, None] * stride_bk + offs_n[None, :] * stride_bn
```

```
for k in range(0, K, BLOCK_SIZE_K):
    mask_k = k + offs_k < K  # Mask for valid K indices
    a = tl.load(a_ptrs, mask=mask_k[None, :], other=0.)  # Load a block of A to shared memory
    b = tl.load(b_ptrs, mask=mask_k[:, None], other=0.)  # Load a block of B to shared memory
    accumulator = tl.dot(a, b, accumulator)  # Call tensor core
    a_ptrs += BLOCK_SIZE_K * stride_ak  # Move to the next block of A
    b_ptrs += BLOCK_SIZE_K * stride_bk  # Move to the next block of B
```

```
mid = (K // BLOCK_SIZE_K) * BLOCK_SIZE_K
for k in range(0, mid, BLOCK_SIZE_K):
    a = tl.load(a_ptrs)  # Load a block of A to shared memory
    b = tl.load(b_ptrs)  # Load a block of B to shared memory
    accumulator = tl.dot(a, b, accumulator)  # Call tensor core
    a_ptrs += BLOCK_SIZE_K * stride_ak  # Move to the next block of A
    b_ptrs += BLOCK_SIZE_K * stride_bk  # Move to the next block of B
if mid < K:
    mask_k = mid + offs_k < K  # Mask for valid K indices
    a = tl.load(a_ptrs, mask=mask_k[None, :], other=0.)  # Load a block of A to shared memory
    b = tl.load(b_ptrs, mask=mask_k[:, None], other=0.)  # Load a block of B to shared memory
    accumulator = tl.dot(a, b, accumulator)  # Call tensor core
```

Reduce ALU & Interger Instructions

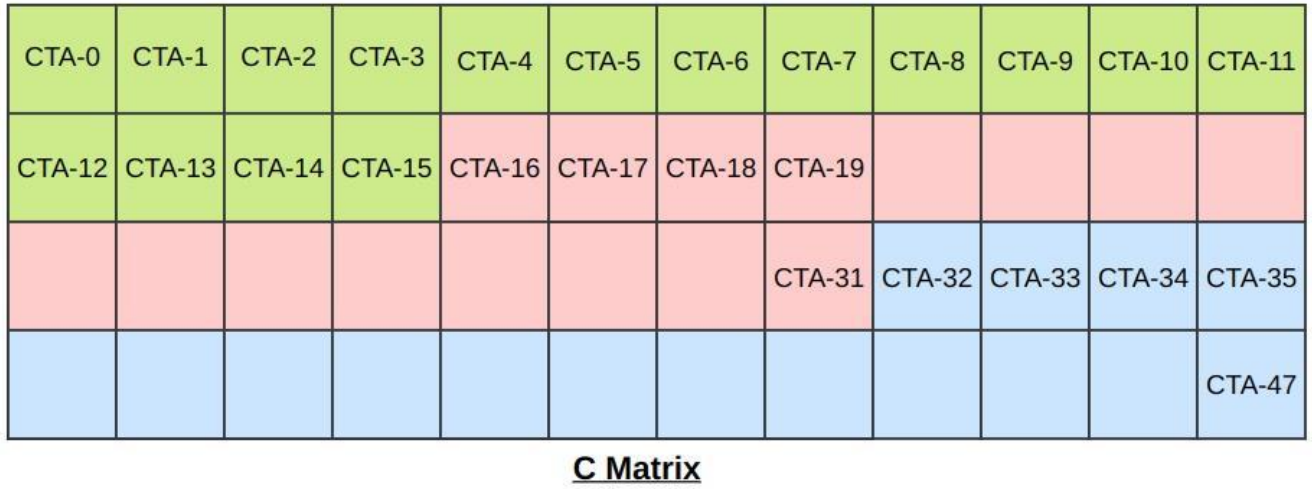
- Not ideal

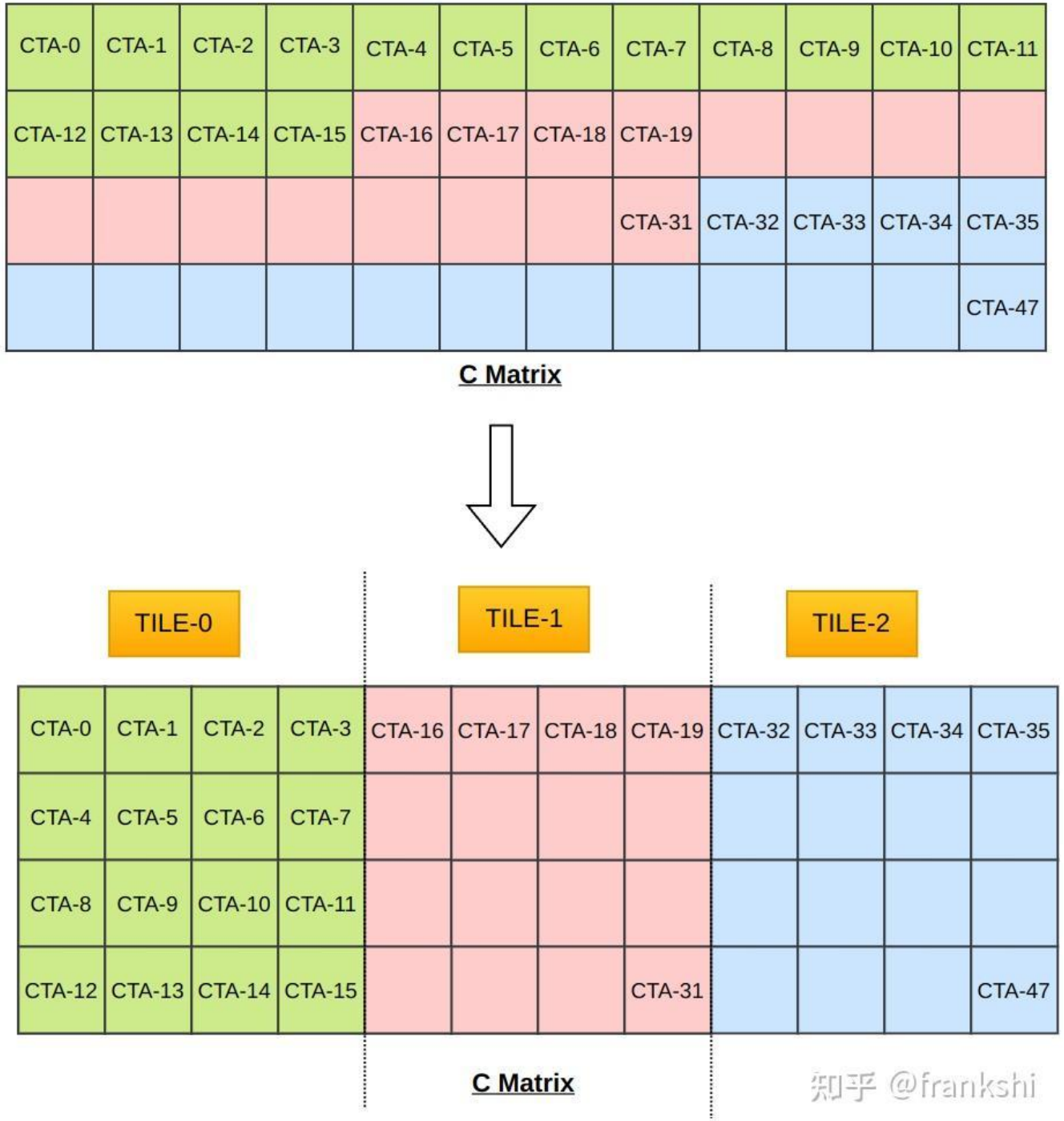
```
Problem size: (4096, 4096, 4096)
Torch GeMM Latency: 0.632 ms
Triton (Naive) GeMM Latency: 0.679 ms
Triton (Good) GeMM Latency: 0.684 ms
```

- Reason: Limited Registers

Function Name	Demangled Name	Duration [us] (6,242.37 us)	Runtime Improvement [us] (1,571.32 us)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread
ampere_fp16_s168...	ampere_fp16_s168...	667.26 (-12.74%)	133.45	87.92 (+13.92%)	47.33 (+12.95%)	218 (-14.51%)
matmul_kernel	**matmul_kernel**	764.67	174.52	77.18	41.90	255
matmul_kernel	matmul_kernel	772.90 (+1.08%)	182.94	76.33 (-1.10%)	41.43 (-1.13%)	253 (-0.78%)

Threadblock Swizzle



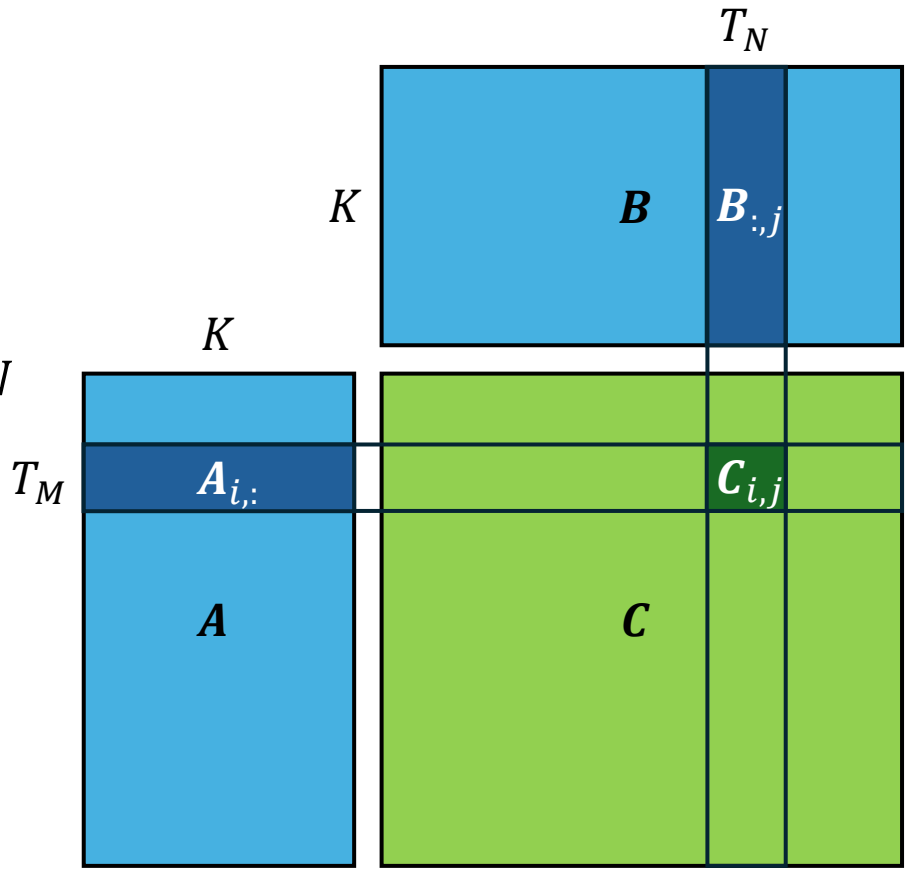

CTA-0 CTA-1 CTA-2 CTA-3 CTA-4 CTA-5 CTA-6 CTA-7 CTA-8 CTA-9 CTA-10 CTA-11
CTA-12 CTA-13 CTA-14 CTA-15 CTA-16 CTA-17 CTA-18 CTA-19
CTA-31 CTA-32 CTA-33 CTA-34 CTA-35
CTA-47
C Matrix
TILE-0
TILE-1
TILE-2
CTA-0 CTA-1 CTA-2 CTA-3 CTA-16 CTA-17 CTA-18 CTA-19 CTA-32 CTA-33 CTA-34 CTA-35
CTA-4 CTA-5 CTA-6 CTA-7
CTA-8 CTA-9 CTA-10 CTA-11
CTA-12 CTA-13 CTA-14 CTA-15 CTA-31 CTA-47
C Matrix
知乎 @frankshi

Arithmetic Intensity of GeMM

- Block size $[T_M, T_N] \ll K$
- $\boldsymbol{C}_{ij} = \sum_{k=0}^{K} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj}$
- Block HBM Access: $T_M K + T_N K + T_M T_N$
- Block Calculation: $T_M T_N K$
- Arithmetic Intensity :

$$\frac{T_M T_N K}{T_M K + T_N K + T_M T_N} \approx \frac{T_M T_N}{T_M + T_N}$$

- $\frac{128\times256}{128+256} > \frac{128\times128}{128+128}$



Change Block Size & Threadblock Swizzle

```
BLOCK_SIZE_M = 128
BLOCK_SIZE_N = 256
BLOCK_SIZE_K = 64
grid = (triton.cdiv(N, BLOCK_SIZE_N), triton.cdiv(M, BLOCK_SIZE_M), )
matmul_kernel[grid](
    a, b, c, M, N, K,
    a.stride(0), a.stride(1),
    b.stride(0), b.stride(1),
    c.stride(0), c.stride(1),
    num_stages=3, num_warps=8,
```

(Compute bound)

(Balanced)

```
BLOCK_SIZE_M = 128
BLOCK_SIZE_N = 128
BLOCK_SIZE_K = 32
grid = (triton.cdiv(N, BLOCK_SIZE_N) * triton.cdiv(M, BLOCK_SIZE_M), )
matmul_kernel[grid](
    a, b, c, M, N, K,
    a.stride(0), a.stride(1),
    b.stride(0), b.stride(1),
    c.stride(0), c.stride(1),
    num_stages=4, num_warps=4,
```

```
# Thread block index
pid_n = tl.program_id(axis=0)
pid_m = tl.program_id(axis=1)
start_m = pid_m * BLOCK_SIZE_M
start_n = pid_n * BLOCK_SIZE_N
```

```
# Thread block index
GROUP_SIZE_M = 8
pid = tl.program_id(axis=0)
num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
num_pid_in_group = GROUP_SIZE_M * num_pid_n
group_id = pid // num_pid_in_group
first_pid_m = group_id * GROUP_SIZE_M
group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
pid_n = (pid % num_pid_in_group) // group_size_m
start_m = pid_m * BLOCK_SIZE_M
start_n = pid_n * BLOCK_SIZE_N
```

Change Block Size & Threadblock Swizzle

- Good

```
Problem size: (4096, 4096, 4096)
Torch GeMM Latency: 0.632 ms
Triton (Naive) GeMM Latency: 0.679 ms
Triton (Good) GeMM Latency: 0.684 ms
Triton (Better) GeMM Latency: 0.665 ms
Problem size: (4321, 4080, 4080)
Torch GeMM Latency: 0.729 ms
Triton (Naive) GeMM Latency: 0.792 ms
Triton (Good) GeMM Latency: 0.819 ms
Triton (Better) GeMM Latency: 0.733 ms
```

Reading Materials: Nsight Compute

- 4. Nsight Compute CLI — NsightCompute 13.0 documentation
- NVIDIA Development Tools Solutions - ERR_NVGPUCTRPERM: Permission issue with Performance Counters | NVIDIA Developer

Homework

- Play with TritonStudyGroup/4_Nsight_Compute at main · Starmys/TritonStudyGroup

Reading Materials: Triton Static Hints

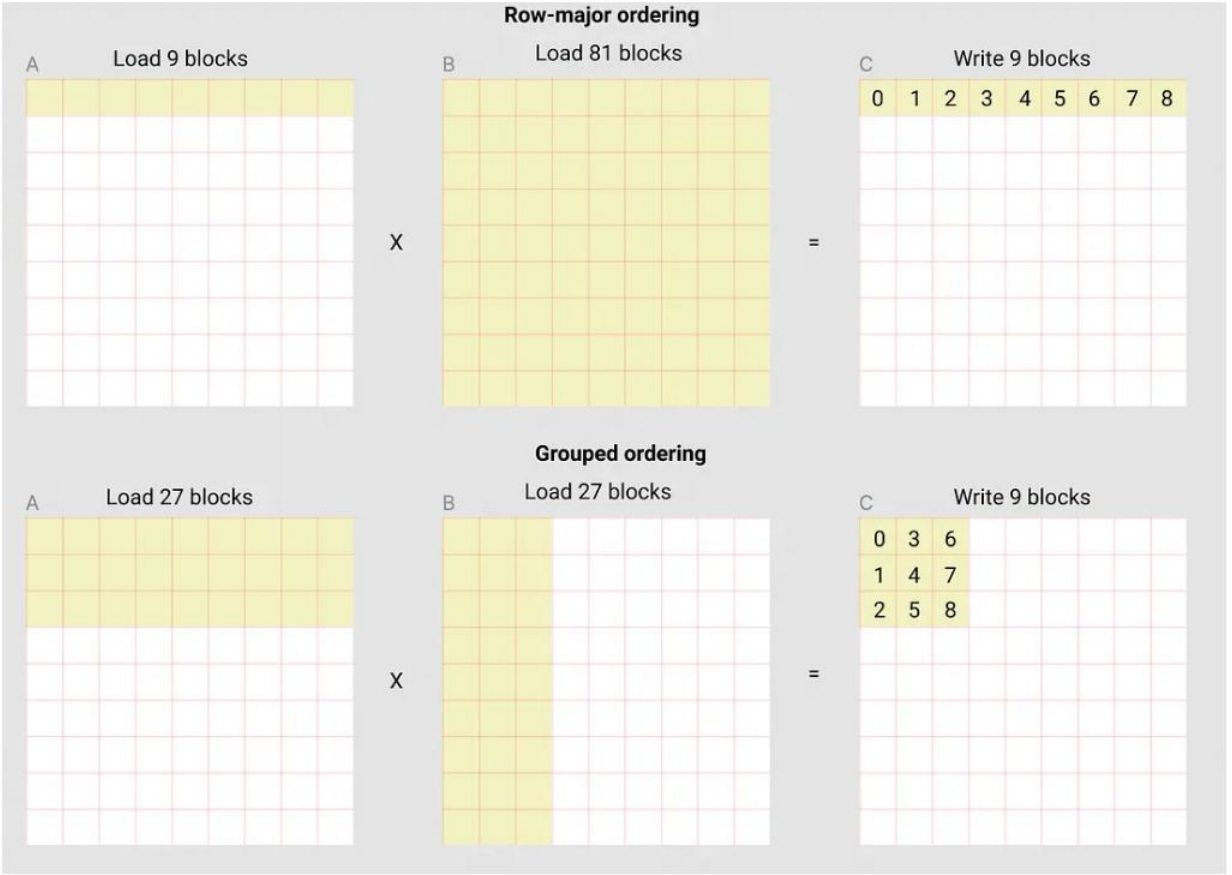
- Confusion of arange, multiple_of, max_contiguous · triton-lang/triton · Discussion #5939
- triton/include/triton/Analysis/AxisInfo.h at 65d9862f5a9029827a7ae04439f07d7e7eadf859 · triton-lang/triton

Reading Materials

- Understanding the Triton Tutorials Part 1 | by Isamu Isozaki | Medium
- Deep Dive on CUTLASS Ping-Pong GEMM Kernel – PyTorch

L2 Cache Optimizations

One way the tutorial suggests we do this is very simple, just increase the number of rows loaded at once!



As you see, if we load one row and load all the columns, we can write 9 blocks in the output. But if we load 3 rows and 3 columns, we can write 9 blocks but without loading 81–54 = 27 rows! In practice, this can save around 10% of computing.