Triton on AMD GPUs

Lei Zhang

Lixun Zhang

AMD Triton Team

Agenda

- Features and status
- Selected optimizations

Features and status

AMD backend officially supported since Triton 3.0!

Functionality features

- Various AMD GPU matrix core intrinsics
- Data types: fp32, fp16, bf16, fp8, int8
- Various passes for performant matmul and attention

- Tools: proton support via roctracer
- Slimmer dependencies for releases
  - Device libraries only needs ockl and ocml
  - Reusing torch HIP runtime

- AMD Instinct GPUs and Radeon GPUs ✅

Performance features for MI300

- Compute: selecting intrinsic variants; fast exp math for attention; …
- Memory access:
  - Global load into matrix core layout*
  - Using buffer intrinsics*
- Cache controls: chiplet-friendly PID mapping; cache modifiers…

- Software pipelining: Common pipeliner infrastructure; …
  - More advanced scheme*; …
- Instruction scheduling:
  - LLVM sched_group_barrier for better ILP*

* Work in progress

Matmul performance

	M	N	K	TFLOP/s
fp16	4096	4096	4096	522
fp16	4096	4096	4160	551
fp16	4864	4096	4160	588
fp16	4864	8192	4160	600
fp8	4096	4096	8192	1024
fp8	4864	4096	8192	1111
fp8	4864	4096	8320	1139
fp8	4864	8192	16512	1216

- ROCm/triton perf kernel
- A: row-major
- B: col-major

Attention performance

L / N_CTX	TFLOP/s
1024	322
2048	389
4096	480
8192	492
16384	518

- ROCm/triton perf kernel
- V: [B, H, D, L]
- B=4
- H=48
- D=128

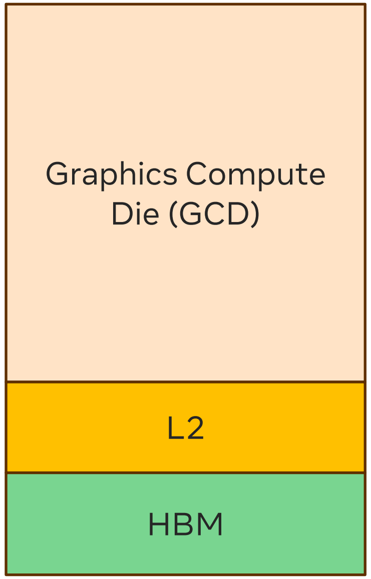
Selected Optimizations

Zooming in on a few optimizations towards MI300:

* Chiplet-friendly PID mapping for L2 locality
* Instruction scheduling for global memory and L1
* Stream-K triton GEMM kernel

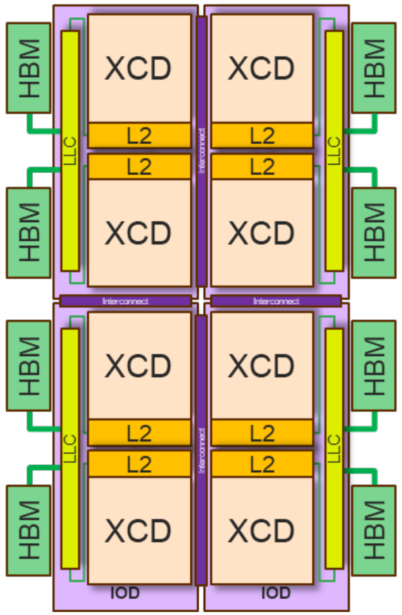
Memory disaggregation

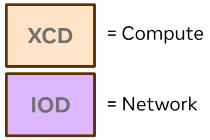
MI200 – Monolithic



Aggregated Memory
No L2 Cache NUMA
No HBM NUMA

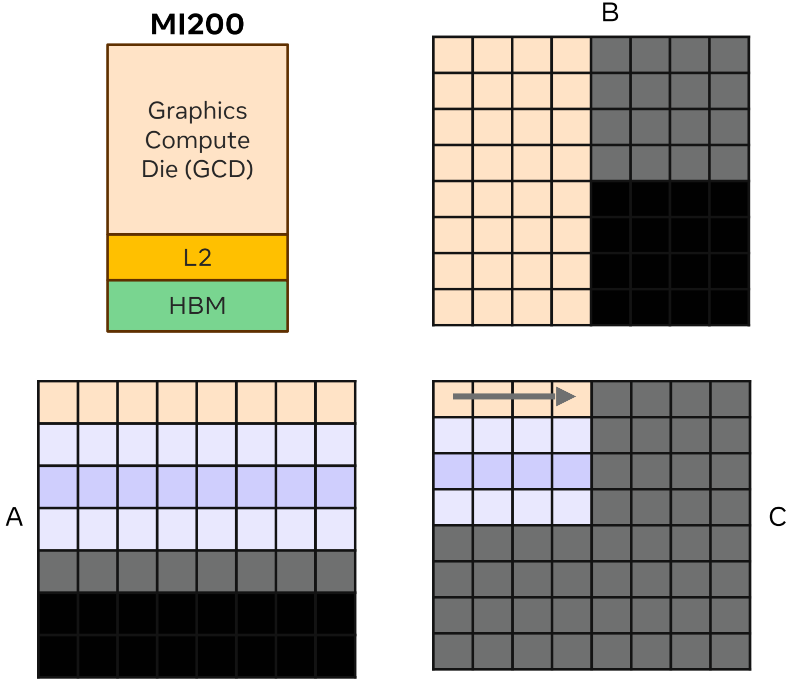
MI300 - Chiplet



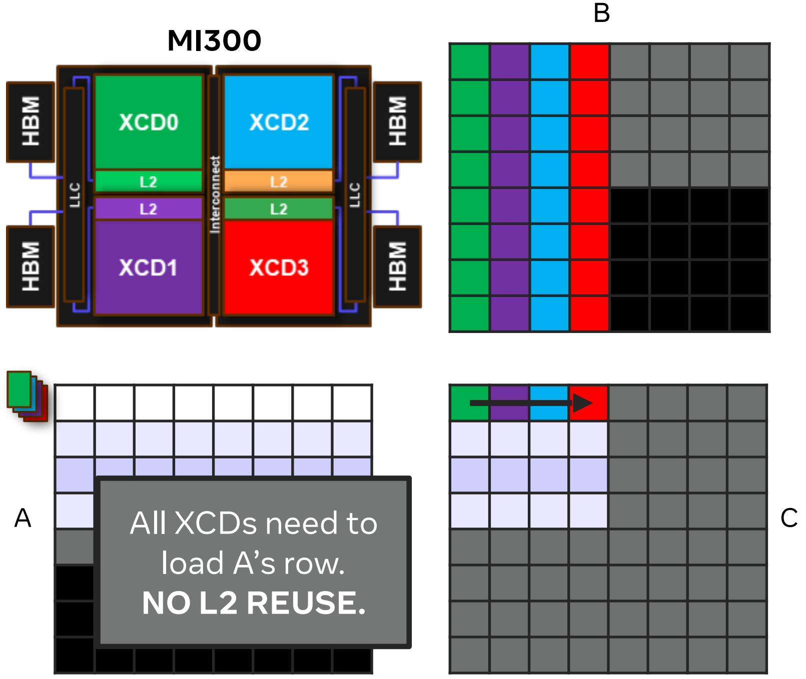


Disaggregated Memory
L2 Cache NUMA
HBM Stack NUMA

Disaggregated L2 loads more tiles with default scheduling



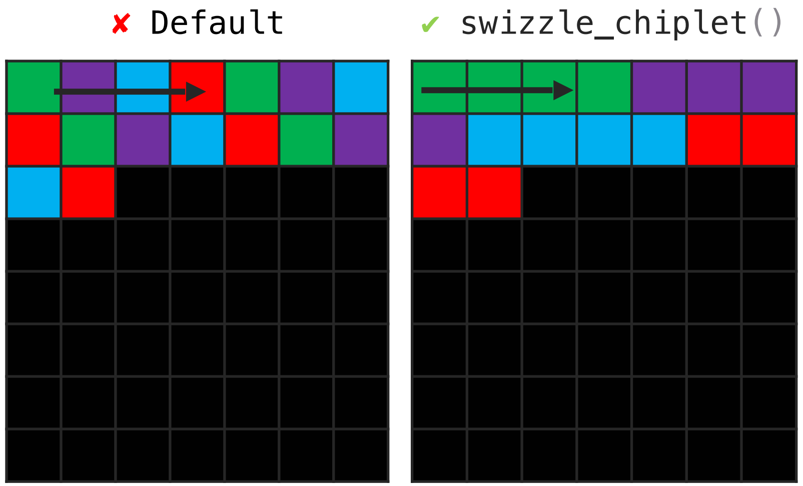
8 A tiles + 32 B tiles = **40 tiles**



All XCDs need to load A's row.
NO L2 REUSE.

(4 XCD * 8 A tiles) + 32 B tiles = **64 tiles**

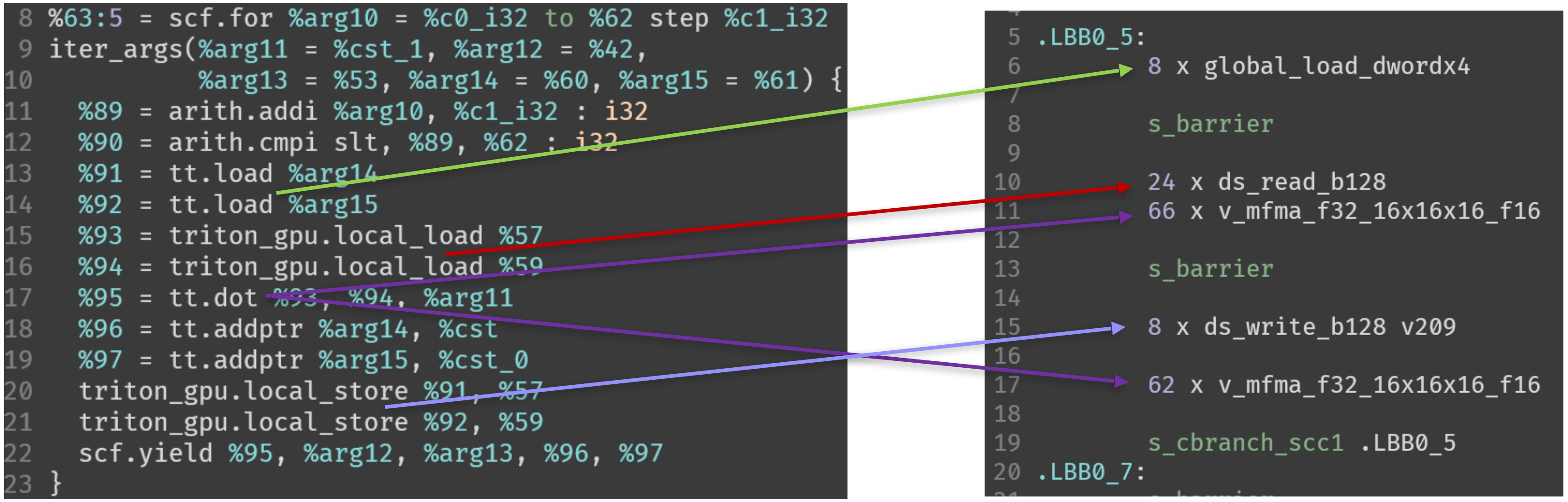
Swizzling to adapt to disaggregated L2 scheduling



- pid_per_xcd = grid // NUM_XCD
- Swizzles the program IDs such that each XCD **gets pid_per_xcd worth of contiguous workgroups**
- Can gain up to 10% perf

Instruction scheduling: default status

ttgir   amdgcn



```
%63:5 = scf.for %arg10 = %c0_i32 to %62 step %c1_i32
iter_args(%arg11 = %cst_1, %arg12 = %42,
          %arg13 = %53, %arg14 = %60, %arg15 = %61) {
  %89 = arith.addi %arg10, %c1_i32 : i32
  %90 = arith.cmpi slt, %89, %62 : i32
  %91 = tt.load %arg14
  %92 = tt.load %arg15
  %93 = triton_gpu.local_load %57
  %94 = triton_gpu.local_load %59
  %95 = tt.dot %93, %94, %arg11
  %96 = tt.addptr %arg14, %cst
  %97 = tt.addptr %arg15, %cst_0
  triton_gpu.local_store %91, %57
  triton_gpu.local_store %92, %59
  scf.yield %95, %arg12, %arg13, %96, %97
}
```

```
.LBB0_5:
        8 x global_load_dwordx4

        s_barrier

        24 x ds_read_b128
        66 x v_mfma_f32_16x16x16_f16

        s_barrier

        8 x ds_write_b128 v209

        62 x v_mfma_f32_16x16x16_f16

        s_cbranch_scc1 .LBB0_5
.LBB0_7:
```

Issuing all global_load_dwordx4 at once is not optimal:

- ✔ Memory latency hidden behind compute
- ✘ Exposed instruction issue latency—memory system congested leading to longer issue latency (100+ cycles)

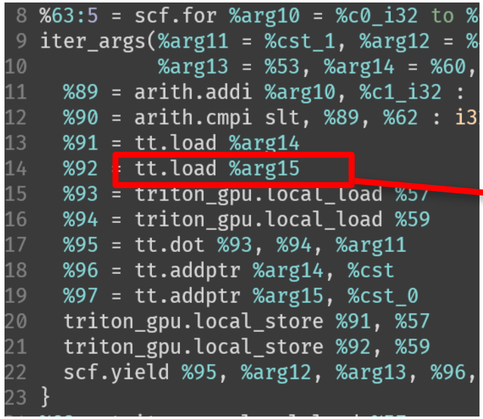
Instruction scheduling: problems and solution

Even worse when K=4096/8192 and BLOCK_K=64

- L1 has 4 sets, and global addresses are linearly mapped
- All global loads map to the same set causing hotspot
- Issue latency becomes even longer, 500+ cycles

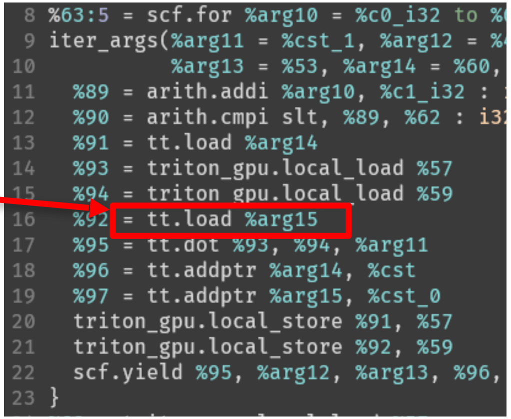
Solution: do not issue all global loads at once

Old Schedule

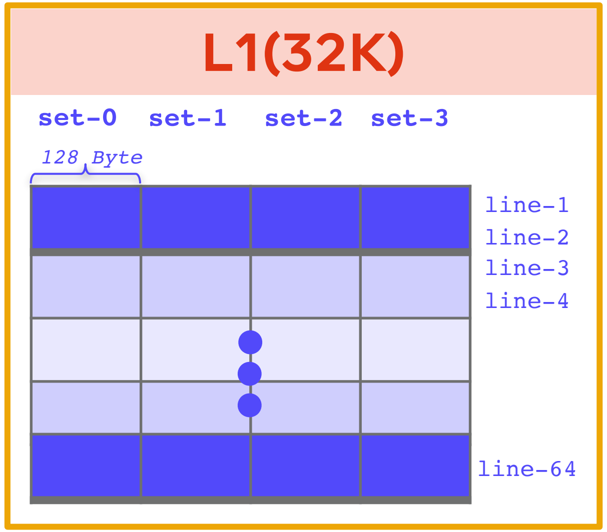


```
%63:5 = scf.for %arg10 = %c0_i32 to %
iter_args(%arg11 = %cst_1, %arg12 = %
          %arg13 = %53, %arg14 = %60,
  %89 = arith.addi %arg10, %c1_i32 :
  %90 = arith.cmpi slt, %89, %62 : i3
  %91 = tt.load %arg14
  %92 = tt.load %arg15
  %93 = triton_gpu.local_load %57
  %94 = triton_gpu.local_load %59
  %95 = tt.dot %93, %94, %arg11
  %96 = tt.addptr %arg14, %cst
  %97 = tt.addptr %arg15, %cst_0
  triton_gpu.local_store %91, %57
  triton_gpu.local_store %92, %59
  scf.yield %95, %arg12, %arg13, %96,
}
```

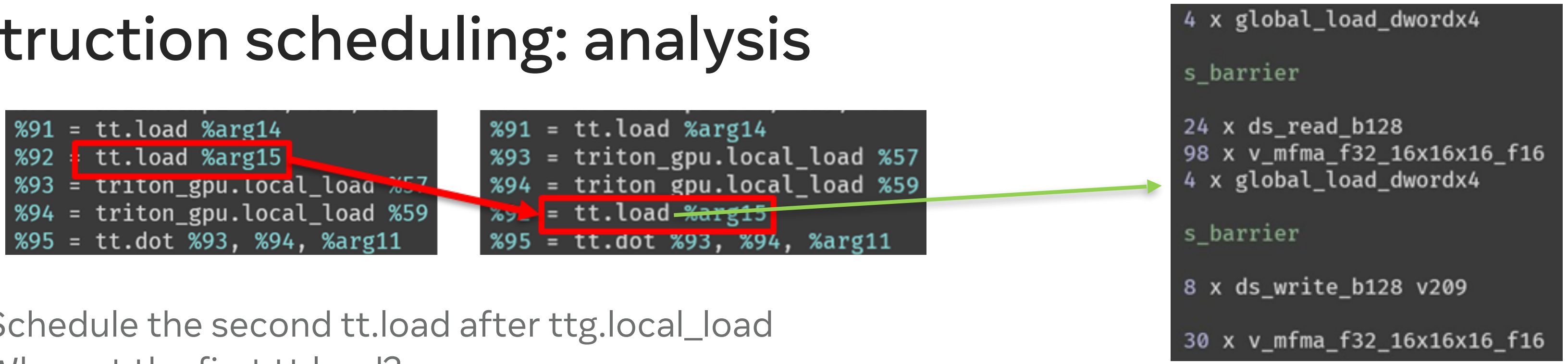
New Schedule



```
%63:5 = scf.for %arg10 = %c0_i32 to %
iter_args(%arg11 = %cst_1, %arg12 = %
          %arg13 = %53, %arg14 = %60,
  %89 = arith.addi %arg10, %c1_i32 :
  %90 = arith.cmpi slt, %89, %62 : i3
  %91 = tt.load %arg14
  %93 = triton_gpu.local_load %57
  %94 = triton_gpu.local_load %59
  %92 = tt.load %arg15
  %95 = tt.dot %93, %94, %arg11
  %96 = tt.addptr %arg14, %cst
  %97 = tt.addptr %arg15, %cst_0
  triton_gpu.local_store %91, %57
  triton_gpu.local_store %92, %59
  scf.yield %95, %arg12, %arg13, %96,
}
```



Instruction scheduling: analysis



```
%91 = tt.load %arg14
%92 = tt.load %arg15
%93 = triton_gpu.local_load %57
%94 = triton_gpu.local_load %59
%95 = tt.dot %93, %94, %arg11
```

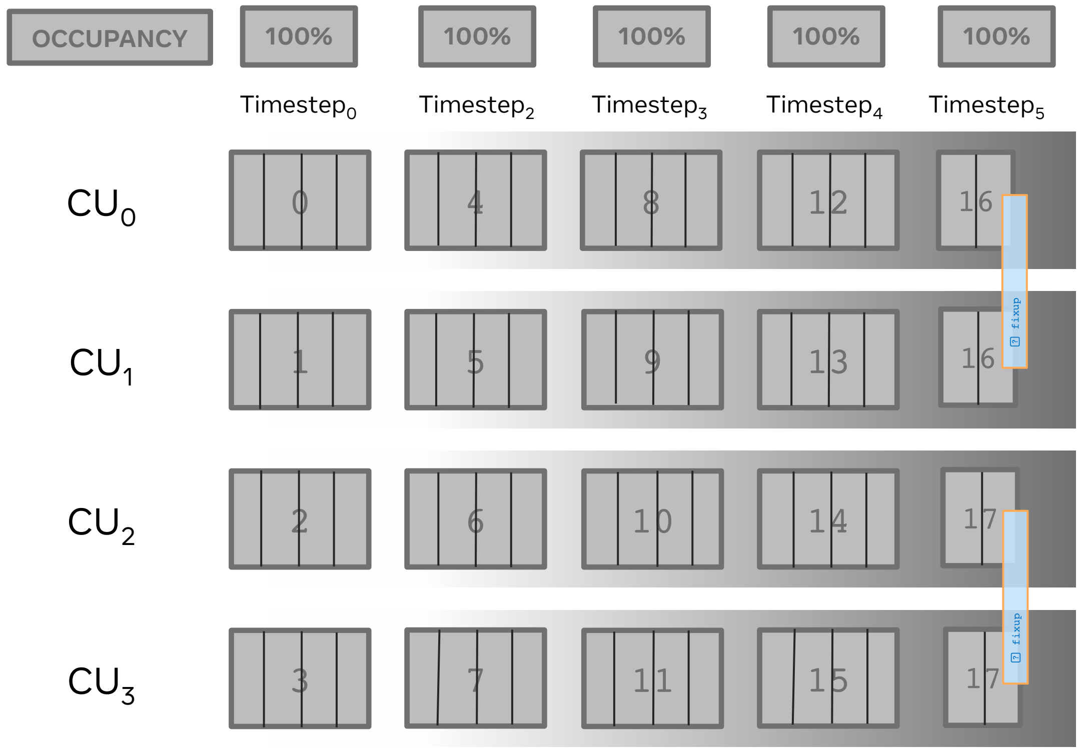
```
%91 = tt.load %arg14
%93 = triton_gpu.local_load %57
%94 = triton_gpu.local_load %59
%92 = tt.load %arg15
%95 = tt.dot %93, %94, %arg11
```

```
4 x global_load_dwordx4

s_barrier

24 x ds_read_b128
98 x v_mfma_f32_16x16x16_f16
4 x global_load_dwordx4

s_barrier

8 x ds_write_b128 v209

30 x v_mfma_f32_16x16x16_f16
```

- Schedule the second tt.load after ttg.local_load
- Why not the first tt.load?
  - Keep the same order of tt.load as their uses, i.e. ttg.local_store
- Why not both tt.load?
  - Backend will push tt.load to the end of mfma; too close to ttg.local_store
  - Collaborating with AMDGPU backend for better **low-level** instruction scheduling

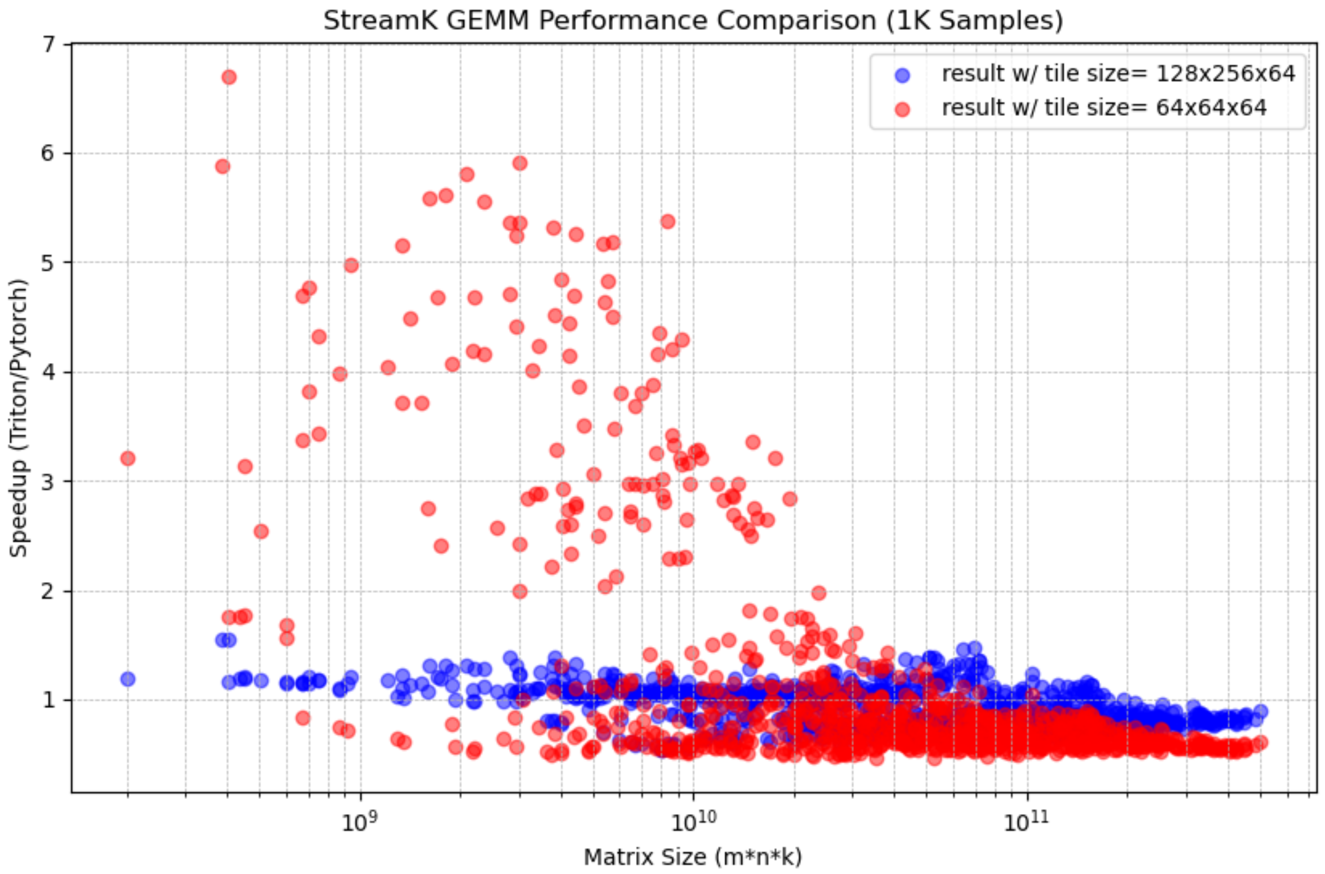
M	N	K	Old	New	Speedup
4096	4096	4096	442	522	**18%**
4864	4096	4096	472	578	**22%**
4096	4096	4160	525	551	**5%**
4864	4096	4160	578	588	**2%**
8192	8192	8192	464	562	**21%**

Stream-K: balanced work decomposition



- Single kernel implementation
- Two tiles approach to reduce synchronization
- Spinning locks to replace atomic_add

Stream-K: performance



Thank you!