Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队
北京大学学生 Linux 俱乐部

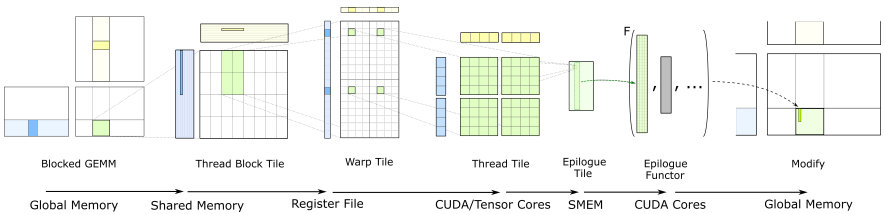
Session 04.
Pipeline Ordering

Yifei Lu

2026. 08. 09

Weiming Supercomputing Team
Linux Club of Peking University

A GEMM's Physical Journey



$$L = \lambda W$$

long latency ⇒ enough work in flight

Tensor Core: high throughput

HBM: long latency

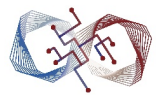
Registers / SMEM: finite

occupancy: often resource-limited

reuse + **overlap**

CUTLASS, Efficient GEMM in CUDA — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/efficient_gemm.html

Baseline Copy–Compute Loop



```
for (size_t batch = 0; batch < batch_sz; ++batch) {
    // Compute the index of the current batch for this block in
global memory:
  size_t block_batch_idx = block.group_index().x * block.size() +
grid.size() * batch;
  size_t global_idx = block_batch_idx + threadIdx.x;
  shared[local_idx] = global_in[global_idx]; //!! COPY

  block.sync(); // Wait for all copies to complete

  compute(global_out + block_batch_idx, shared); // Compute and
write result to global memory

  block.sync(); // Wait for compute using shared memory to finish
  }
```

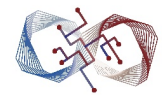
GMEM → register → SMEM
extra instruction + register transit

copy → wait → compute
little copy / compute overlap

second barrier
protects buffer reuse

CUDA Programming Guide, Asynchronous Data Copies — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

The First Proposal: Issue the Next Copy Early



SERIALIZED

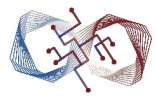
```
copy k → wait → compute k
```

DESIRED

```
issue copy k+1 → compute k → wait
before use
```

A whole block moves one tile. **Which threads belong to the copy protocol?**

Hidden Participants



```
__global__ void kernel(float const* gmem) {
  extern __shared__ float smem[];

  smem[threadIdx.x] = gmem[threadIdx.x];
  __syncthreads();

  consume_together(smem);
}
```

```
if (threadIdx.x < blockDim.x / 2)
  reduce(smem);   // reduce() contains __syncthreads()
```

Whole-block contract

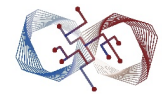
hidden inside `__syncthreads()`

Partial call

missing participants ⇒ deadlock / undefined behavior

NVIDIA Technical Blog, Cooperative Groups — https://developer.nvidia.com/blog/cooperative-groups/

Cooperative Groups



```cpp
#include <cooperative_groups.h>
namespace cg = cooperative_groups;

__global__ void kernel( ... ) {
  cg::thread_block block = cg::this_thread_block();

  int rank = block.thread_rank();
  int size = block.num_threads();

  // ...
  block.sync();
}
```

participant set

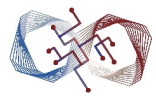
logical rank space

collective contract

```cpp
__device__ void consume_tile(
    cg::thread_block const& block,
    float const* smem) {
  block.sync();
  // consume smem
}
```

CUDA Programming Guide, Cooperative Groups — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html
NVIDIA Technical Blog, Cooperative Groups — https://developer.nvidia.com/blog/cooperative-groups/

Only 32 Threads? Make 32 the Group



wrong scope

```
auto block = cg::this_thread_block();

if (threadIdx.x < 32)
  block.sync();       // block expects everyone
```

partition all parent-group members participate

partition first

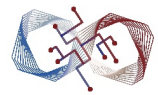
```
auto block = cg::this_thread_block();
auto tile = cg::tiled_partition<32>(block);

if (tile.meta_group_rank() == 0)
  warp_collective(tile);  // all 32 tile members
```

collective all members of the selected tile participate

Cooperative Groups exposes the contract; it does not waive it.

CUDA Programming Guide, Cooperative Groups: Partitioning and Group Creation Hazards — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html

Group Scope Must Match the Algorithm

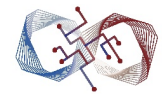


group	participants	typical collective
`thread_block_tile<32>`	one warp-sized tile	`tile.sync()`
`thread_block`	one CTA	`block.sync()`
`cluster_group`	one thread-block cluster	`cluster.sync()`
`grid_group`	the launched grid	`grid.sync()`

`grid.sync()` requires `cudaLaunchCooperativeKernel` and a residency-compatible grid.

CUDA Programming Guide, Cooperative Groups: Large Scale Groups — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html#large-scale-groups

Return to the Async Copy



```
auto block = cg::this_thread_block();

cg::memcpy_async(block, smem_dst, gmem_src, bytes);
do_independent_work();     // cannot read smem_dst
cg::wait(block);

consume(smem_dst);
```

`block`

who cooperates

`memcpy_async`

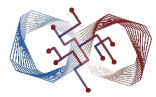
what is issued

`wait`

when data may be consumed

CUDA Programming Guide, Cooperative Groups: Asynchronous Data Movement — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html#asynchronous-data-movement

Three `memcpy_async` Contracts

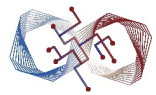


`cg::memcpy_async(group, …)`	group collective	`cg::wait(group)`
`cuda::memcpy_async(dst, src, …, sync)`	calling thread issues one copy	`pipeline` / `barrier`
`cuda::memcpy_async(group, dst, src, …, sync)`	group overload: collective copy	`pipeline` / `barrier`

Namespace, group argument, and completion object together define the contract.

CCCL / libcu++, cuda::memcpy_async API — https://nvidia.github.io/cccl/libcudacxx/extended_api/asynchronous_operations/memcpy_async.html

`wait` Is Not the Whole Correctness Proof

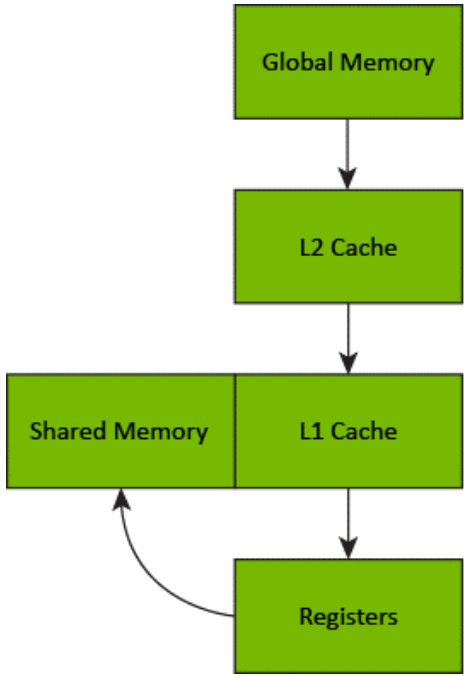


1 · Address and range	valid pointers · in bounds · no overlap
2 · Completion	wait on the group / pipeline / barrier that tracks this copy
3 · Reuse	do not overwrite a stage before its previous consumer releases it

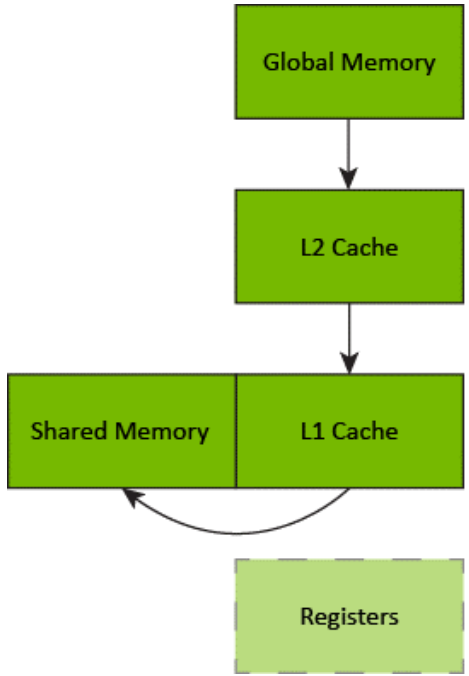
A completed copy can still be the wrong copy, or land in a stage reused too early.

CUDA Programming Guide, Asynchronous Data Copies — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

Ampere/SM80: LDGSTS / `cp.async`



LDG → register → STS

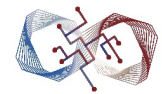


LDGSTS / `cp.async` → SMEM

normal global-memory path and caches; no explicit GPR payload

NVIDIA Technical Blog, Controlling Data Movement on Ampere — https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/
CUDA Programming Guide, Asynchronous Data Copies — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

Performance Conditions



Alignment

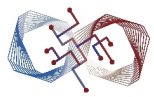
- async path: GMEM → SMEM
- minimum: 4-byte alignment
- recommended: 16-byte alignment
- copy size: compatible granularity

Warp entanglement

- warp-shared batch sequence
- divergent commit ⇒ sequence skew
- over-wait / extra barrier updates
- keep commit and arrive-on converged

CUDA Programming Guide, Asynchronous Data Copies — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

`cuda::pipeline` : an N-stage Queue



producer_acquire → **issue copies** → **producer_commit** → **consumer_wait** → **compute** → **consumer_release**

empty stage → future tile → publish work → oldest stage ready → consume tile → stage reusable

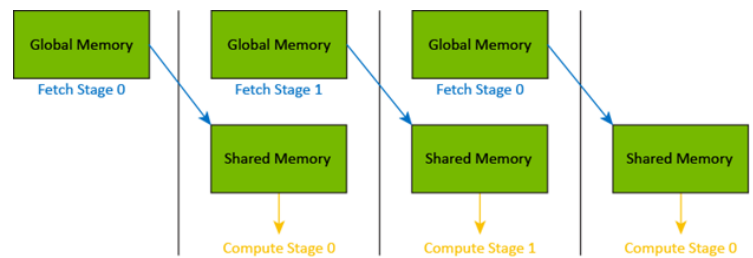
producer owns EMPTY

consumer owns FULL

CUDA Programming Guide, Pipelines — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/pipelines.html

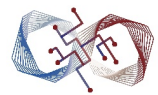
Double Buffering


```
prime(stage[0]);                    // tile 0
for (int k = 0; k < K_tiles - 1; ++k) {
  acquire_empty(stage_for(k + 1));
  issue_copy(stage_for(k + 1), tile(k + 1));
  commit();

  wait_until_full(stage_for(k));
  mma(stage_for(k));                // copy k+1 in flight
  release(stage_for(k));
}
drain_the_final_stage();
```



CUDA Programming Guide, Multi-stage Asynchronous Data Copies — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

Double Buffering Can Still Expose Latency



copy k+1 is still late

transfer latency > compute window

all warps reach the same wait

no ready work inside the CTA

resource limits

too few stages or resident warps

active → **eligible** → **issued**

resident — next instruction ready — selected now

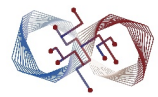
A stalled warp is normal. An empty issue opportunity is the loss.

Occupancy: resident active warps / maximum resident warps.

threads/block, registers/thread, SMEM/block, maximum blocks or warps

Nsight Compute Profiling Guide, Metrics Guide — https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-guide

A Controlled Experiment: One Chain or Two?



one chain

```
unsigned p = tid & mask;

#pragma unroll 1
for (int i = 0; i < iters; ++i){
  p = next[p];
}
out[tid] = p;
```

1 dependent load / iteration

two independent chains

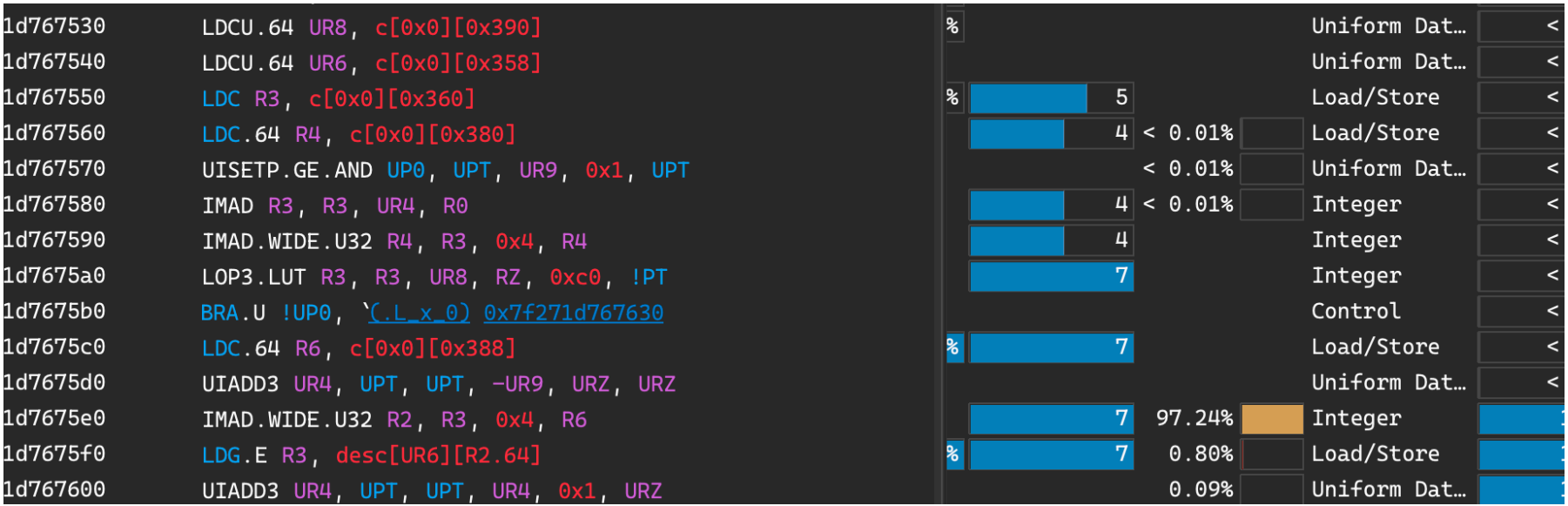
```
unsigned p0 = tid & mask;
unsigned p1 = (p0 + ((mask + 1) >> 1)) & mask;

#pragma unroll 1
for (int i = 0; i < iters; ++i) {
  p0 = next[p0];
  p1 = next[p1];
}

out[tid] = p0 ^ p1;
```

2 dependent loads / iteration

RTX 5090 · SM 12.0 170 blocks × 32 threads same `iters`

Can a second independent chain overlap the first chain's load latency?

One Chain: the Next Address Does Not Exist Yet

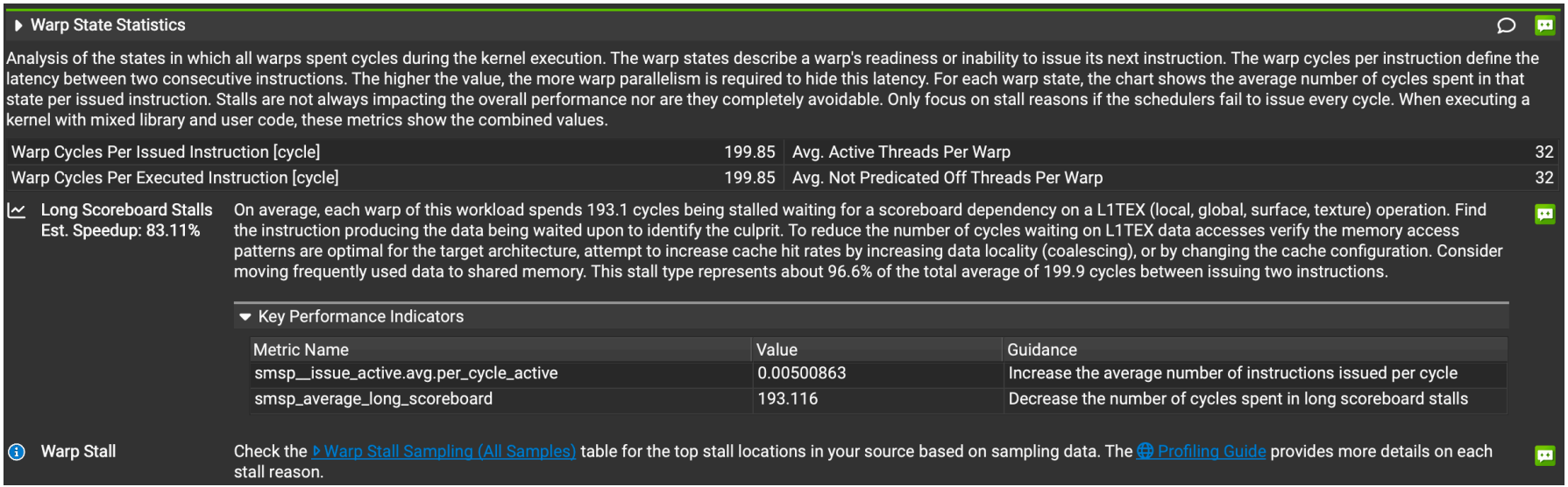


`LDG.E writes R3` ⟶ `next iteration: IMAD.WIDE reads R3`

loop-carried RAW dependency `IMAD.WIDE` is the sampled blocked PC; `LDG.E` is the producer.

Nsight Compute 2025.3, Source Page — https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#source-page

Nsight Names the Wait: Long Scoreboard



▸ 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]	199.85	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	199.85	Avg. Not Predicated Off Threads Per Warp	32

Long Scoreboard Stalls
Est. Speedup: 83.11%

On average, each warp of this workload spends 193.1 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 96.6% of the total average of 199.9 cycles between issuing two instructions.

▾ Key Performance Indicators

Metric Name	Value	Guidance
smsp__issue_active.avg.per_cycle_active	0.00500863	Increase the average number of instructions issued per cycle
smsp_average_long_scoreboard	193.116	Decrease the number of cycles spent in long scoreboard stalls

Warp Stall

Check the ▸ Warp Stall Sampling (All Samples) table for the top stall locations in your source based on sampling data. The Profiling Guide provides more details on each stall reason.

199.85 cycles per issued instruction

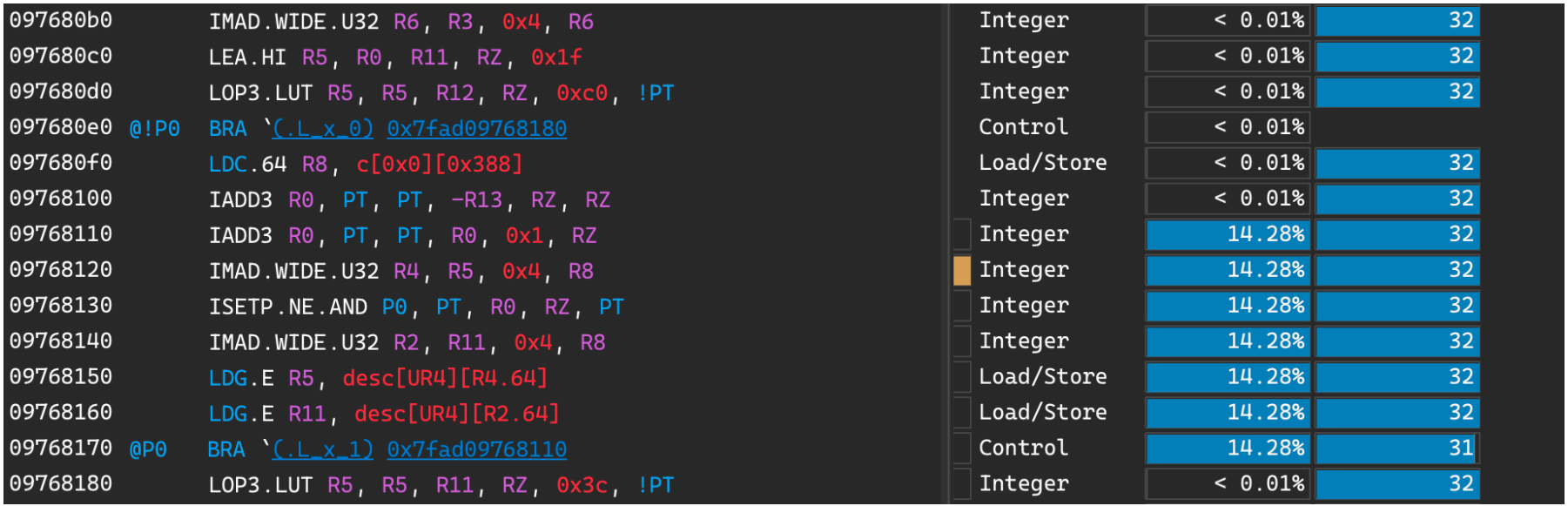
193.1 cycles Long Scoreboard

96.6% of the issue interval

The warp is waiting for a scoreboard dependency produced through L1TEX.

Nsight Compute, Warp State Statistics and Warp Stall Reasons — https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

Two Chains Put Two Loads in Flight



```
097680b0        IMAD.WIDE.U32 R6, R3, 0x4, R6
097680c0        LEA.HI R5, R0, R11, RZ, 0x1f
097680d0        LOP3.LUT R5, R5, R12, RZ, 0xc0, !PT
097680e0 @!P0   BRA `(.L_x_0) 0x7fad09768180
097680f0        LDC.64 R8, c[0x0][0x388]
09768100        IADD3 R0, PT, PT, -R13, RZ, RZ
09768110        IADD3 R0, PT, PT, R0, 0x1, RZ
09768120        IMAD.WIDE.U32 R4, R5, 0x4, R8
09768130        ISETP.NE.AND P0, PT, R0, RZ, PT
09768140        IMAD.WIDE.U32 R2, R11, 0x4, R8
09768150        LDG.E R5, desc[UR4][R4.64]
09768160        LDG.E R11, desc[UR4][R2.64]
09768170 @P0    BRA `(.L_x_1) 0x7fad09768110
09768180        LOP3.LUT R5, R5, R11, RZ, 0x3c, !PT
```

Integer	< 0.01%	32
Integer	< 0.01%	32
Integer	< 0.01%	32
Control	< 0.01%	
Load/Store	< 0.01%	32
Integer	< 0.01%	32
Integer	14.28%	32
Integer	14.28%	32
Integer	14.28%	32
Integer	14.28%	32
Load/Store	14.28%	32
Load/Store	14.28%	32
Control	14.28%	31
Integer	< 0.01%	32

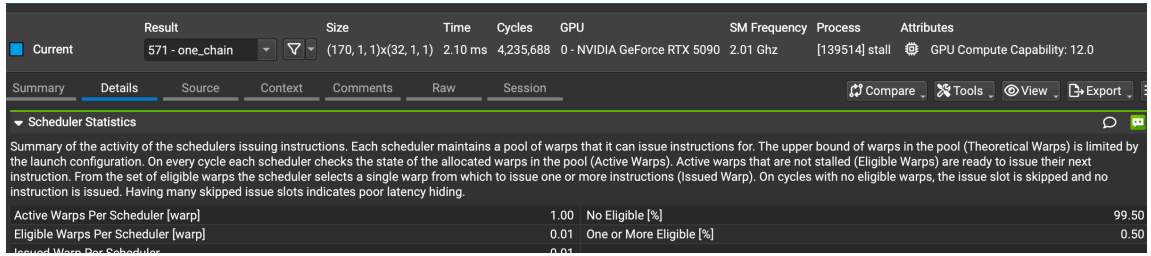
`R5 → address R4 → LDG.E → R5`

`R11 → address R2 → LDG.E → R11`

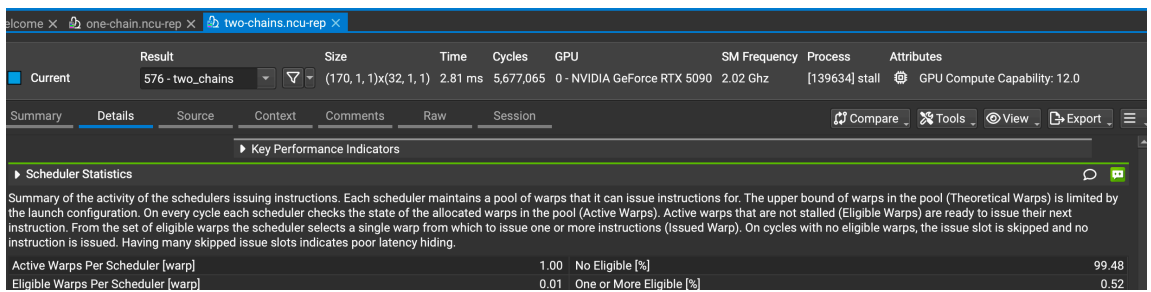
The chains are independent, so the two `LDG.E` instructions can be outstanding together.

Active Does Not Mean Eligible

one chain



two chains



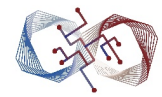
1.00
active warp
per scheduler

0.01
eligible warp
per scheduler

≈99.5%
cycles with
no eligible warp

Measured on NVIDIA GeForce RTX 5090 with Nsight Compute 2025.3

Twice the Loads in 1.34× the Time



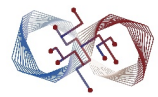
one chain	1 load / iteration	**2.10 ms**
two chains	2 loads / iteration	**2.81 ms**

NORMALIZED LOAD THROUGHPUT

(2 × 2.10) / 2.81 = 1.49×

Raw runtime is not the comparison: the amount of work changed.

Where Can the Missing Eligible Work Come From?



ILP	another instruction chain in the same warp	two pointer chains
MLP	more memory requests outstanding	prefetch · more stages
TLP	another resident warp is ready	more warps / blocks

`two independent chains` → `per-warp ILP` → `more MLP`

Two chains improve load throughput, but they still leave a long window with no eligible work.

CUDA C++ Best Practices Guide, Thread and Block Heuristics — https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#thread-and-block-heuristics

Pipeline Ordering: Four Different Questions

question	meaning
Completion	Has the asynchronous operation finished?
Visibility	May this consumer observe the produced data?
Ordering	Are accesses through different agents/proxies ordered?
Ownership	May another producer reuse this storage?

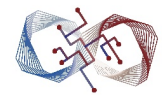
More stages

- earlier prefetch
- longer latency budget

Trade-offs

- more SMEM
- possibly lower occupancy
- more control state

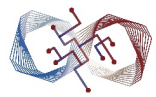
SM80 Takeaway



SM80 improves	you still design
small async GMEM → SMEM path	group participation and addressing
less register transit	alignment and SMEM layout
copy / compute overlap	stage ownership and depth
latency hiding inside a CTA	SMEM / register / occupancy trade-off

Next: reduce the producer's instruction and address-generation work.

Hopper/SM90: TMA



SM80

`many threads → many small copies`

SM90

`one elected issuer → bulk tensor transfer`

Tensor map

source + tile layout

TMA instruction

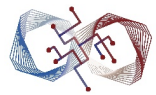
launch one bulk transfer

hardware

address generation + movement

NVIDIA Hopper Architecture In-Depth — https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
CUDA Programming Guide, TMA — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

Tensor Map: the Transfer Contract



```
CUtensorMap tensor_map{};
cuTensorMapEncodeTiled(
  &tensor_map,
  data_type,
  rank,
  tensor_ptr,
  global_dim,
  global_stride,
  box_dim,
  element_stride,
  interleave,
  swizzle,
  l2_promotion,
  oob_fill);
```

Source geometry

base, rank, dimensions, byte strides

Tile geometry

box dimensions, element strides

SMEM placement

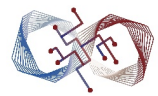
interleave, swizzle

Boundary / cache policy

OOB fill, L2 fetch granularity

CUDA Driver API, Tensor Map Object Management — https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__TENSOR__MEMORY.html

TMA Needs a Completion Object: `mbarrier`



```cpp
using barrier = cuda::barrier<cuda::thread_scope_block>;

__shared__ barrier bar;
if (threadIdx.x == 0)
  init(&bar, blockDim.x);
__syncthreads();

if (is_elected())
  cuda::memcpy_async(smem, gmem,
      cuda::aligned_size_t<16>(bytes), bar);

auto token = bar.arrive();
bar.wait(std::move(token));
consume(smem);
```

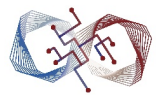
thread arrivals
participants reached the phase

transaction completion
promised bytes arrived

phase flip
stage is FULL

CUDA Programming Guide, TMA bulk asynchronous copy example — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

One Barrier Object, Many Phases



```cpp
// High level: the token names this phase.
for ( ... ) {
  auto token = bar.arrive();
  do_independent_work();
  bar.wait(std::move(token));
  consume_this_tile();
}

// Low level: track phase parity explicitly.
auto native = cuda::device::barrier_native_handle(bar);
int parity = 0;
for ( ... ) {
  (void)cuda::ptx::mbarrier_arrive(native);
  while (!cuda::ptx::mbarrier_try_wait_parity(
      native, parity)) {}
  parity ^= 1;
}
```

Phase 0
wait for parity 0 to complete

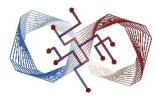
Phase 1
wait for parity 1 to complete

Phase 2
parity returns to 0

The barrier address is reused; the token or parity identifies the generation.

CUDA Programming Guide, Asynchronous Barriers — Explicit Phase Tracking — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-barriers.html

Same SMEM, Two Access Paths



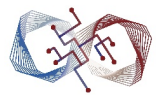
GENERIC PROXY	ASYNC PROXY
`normal CUDA ld / st`	`TMA · WGMMA · tcgen05`

`generic SMEM stores` → `fence_proxy_async(shared)` → `TMA reads SMEM`

Same address does not imply cross-proxy ordering.

CUDA Programming Guide, Async Thread and Async Proxy — https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html

TMA Round Trip: Three Different Edges



```cpp
if (is_elected()) {
  cuda::memcpy_async(smem_data, gmem_src,
      cuda::aligned_size_t<16>(bytes), bar);
}
auto token = bar.arrive();
bar.wait(std::move(token));
consume(smem_data);
```

```cpp
for (int i = threadIdx.x; i < count; i += blockDim.x) {
  smem_data[i] += 1;
}
ptx::fence_proxy_async(ptx::space_shared);
__syncthreads();
```

```cpp
if (is_elected()) {
  ptx::cp_async_bulk(
      ptx::space_global, ptx::space_shared,
      gmem_dst, smem_data, bytes);
  ptx::cp_async_bulk_commit_group();
  ptx::cp_async_bulk_wait_group_read(ptx::n32_t<0>());
}
```

① data ready

`mbarrier` completion

② cross-proxy visibility

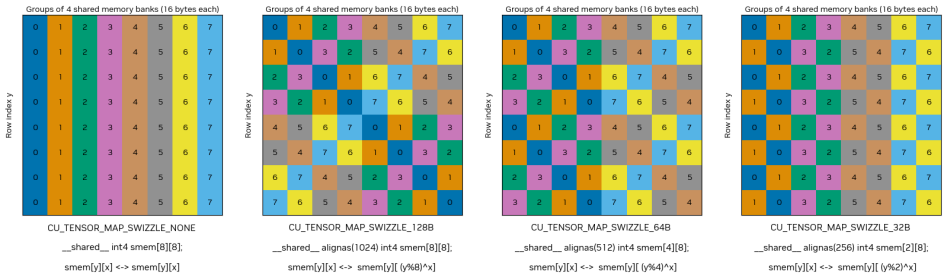
`fence.proxy.async`

③ source reusable

bulk async-group completion

CUDA Programming Guide, TMA one-dimensional read-modify-write example — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html
PTX ISA, Proxy Fence Instructions — https://docs.nvidia.com/cuda/parallel-thread-execution/contents.html

TMA Swizzle: Layout for the Consumer



Swizzle Mode	Offset Formula	Index Relation
CU_TENSOR_MAP_SWIZZLE_128B	`(reinterpret_cast <uintptr_t>(smem_ptr)/128)%8`	`smem[y][x] <-> smem[y][((y+offset)%8)^x]`
CU_TENSOR_MAP_SWIZZLE_64B	`(reinterpret_cast <uintptr_t>(smem_ptr)/128)%4`	`smem[y][x] <-> smem[y][((y+offset)%4)^x]`
CU_TENSOR_MAP_SWIZZLE_32B	`(reinterpret_cast <uintptr_t>(smem_ptr)/128)%2`	`smem[y][x] <-> smem[y][((y+offset)%2)^x]`

default: preserve tensor order

swizzle: change SMEM bank placement

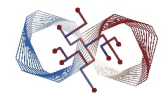
goal: fewer bank conflicts

consumer access pattern decides the layout

GMEM order ≠ best SMEM order

CUDA Programming Guide, TMA Swizzle — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

TMA Swizzle: Mode-specific Constraints



Alignment

base TMA · swizzle mode · SMEM base

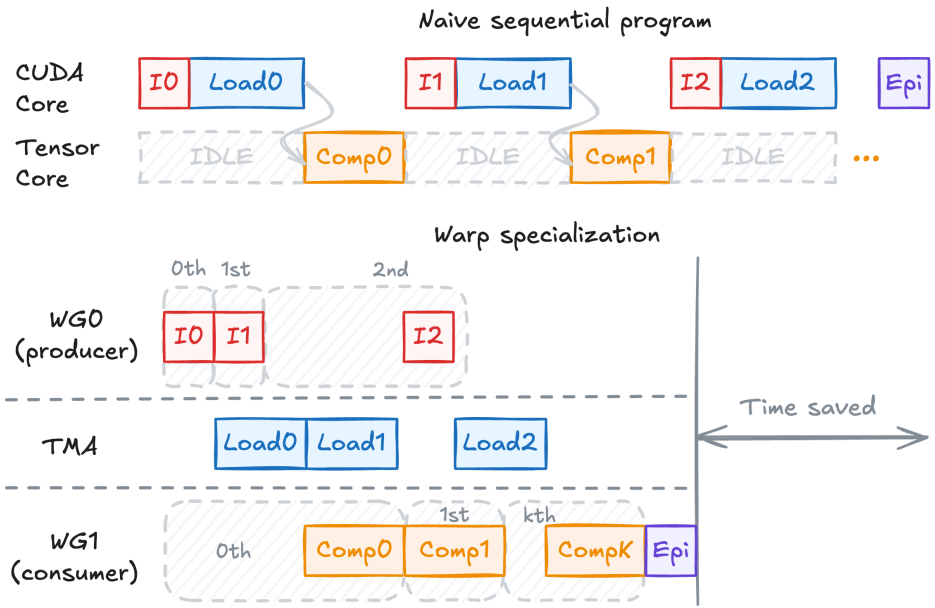
Granularity

16-byte access units

Pattern	Swizzle width	Shared box's inner dimension	Repeats after	Shared memory alignment	Global memory alignment
CU_TENSOR_MAP_SWIZZLE_128B	128 bytes	<=128 bytes	1024 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZLE_64B	64 bytes	<=64 bytes	512 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZLE_32B	32 bytes	<=32 bytes	256 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZLE_NONE (default)				128 bytes	16 bytes

CUDA Programming Guide, TMA Swizzle Considerations — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/async-copies.html

Warp Specialization: Divide the CTA into Agents



producer warp group

- TMA issue
- empty / full stages

consumer warp group

- WGMMA
- accumulators
- stage release

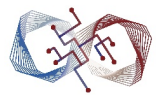
TMA = transfer mechanism · multistage buffering = schedule

CUTLASS, Hopper Warp Specialization — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/efficient_gemm.html

SM90 Takeaway

The same ownership cycle; a different producer and completion mechanism.

Blackwell/SM100: Why TMEM?



SM80 / SM90

`Tensor Core → register accumulator`

SM100

`TCGEN05`

`Tensor Core → TMEM accumulator`

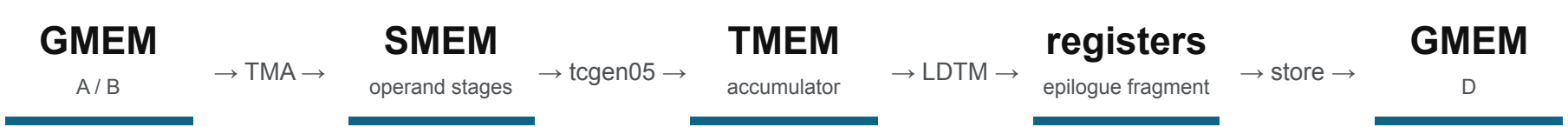
SMEM
A / B operand stages

TMEM
accumulator lifetime

registers
control + epilogue fragments

CUTLASS, tcgen05 MMA Programming Guide — https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/mma_docs/tcgen05_programming.html

TMEM Dataflow



`tAgA / tBgB` → `tAsA / tBsB` → `tCtAcc` → `tTR_rAcc` → `tTR_gC`

CUTLASS, tcgen05 GMEM-to-MMA Dataflow — https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/mma_docs/tcgen05_programming.html

tcgen05 : Quick Recall

one elected thread issues MMA

A: SMEM or TMEM

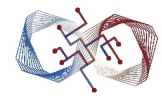
B: SMEM

accumulator: TMEM

optional 2-CTA cooperation

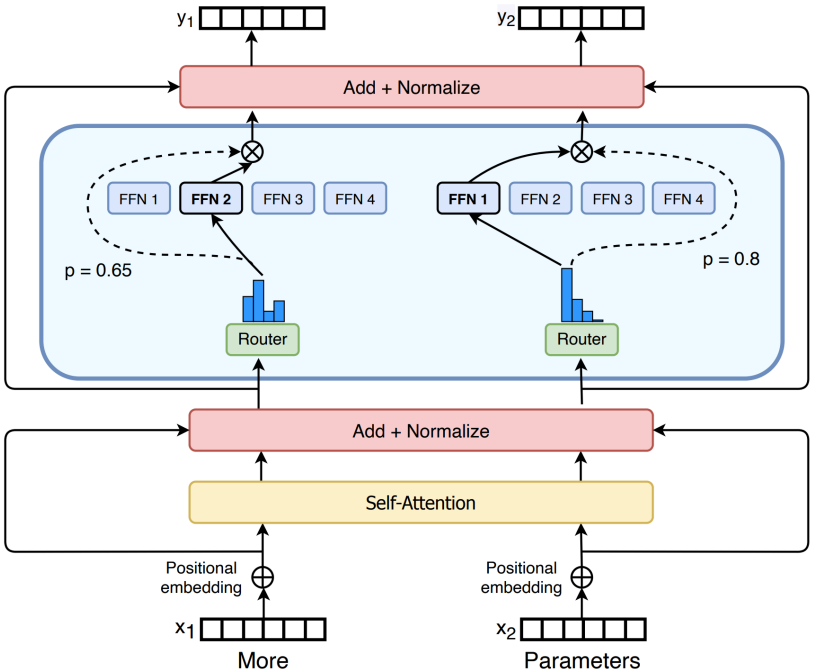
CUTLASS, tcgen05 MMA Programming Guide — https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/mma_docs/tcgen05_programming.html

SM100 Takeaway



generation	new capability	new orchestration task
SM80	small async GMEM → SMEM	copy stages
SM90	TMA + mbarrier + WGMMA	bulk transfer + warp roles
SM100	TMEM + 2-CTA `tcgen05`	accumulator + CTA-pair lifetime

A Good Tile Is Not Enough: MoE



tokens route to experts

experts receive different token counts

many GEMMs with different `M`

unequal tile counts and durations

irregular work ⇒ tail imbalance

Switch Transformers — https://arxiv.org/abs/2101.03961

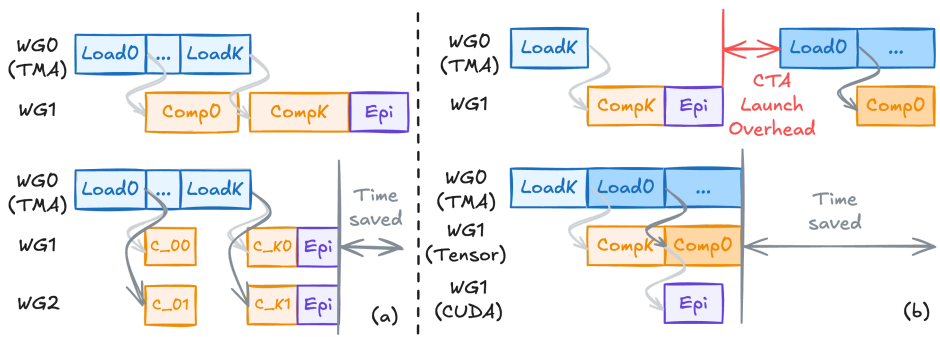
Persistent Kernel: Keep a Worker Alive

CONVENTIONAL TILED GEMM

`CTA → one output tile → exit`

PERSISTENT GEMM

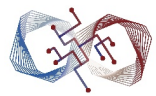
`CTA → tile 0 → tile 1 → tile 2 → exit`



- bounded worker grid
- scheduler returns next tile
- setup amortized across tiles
- inner TMA ↔ MMA pipeline repeats

CUTLASS, Persistent Cooperative Kernels — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/efficient_gemm.html

The Persistent Outer Loop



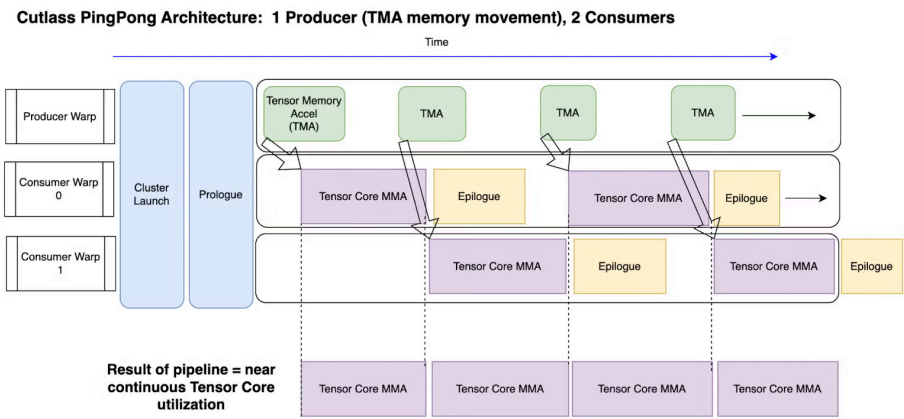
```
WorkerState state = initialize_once();

while (scheduler.next_tile(work)) {
  bind_problem_and_coordinates(state, work);
  compute_one_output_tile(state, work);
  finish_epilogue_and_release(state, work);
}

drain_and_exit(state);
```

per worker
roles, barriers, scheduler state

per stage
SMEM slot + phase

per tile
coordinates, pointers, epilogue

Persistent Ping-Pong



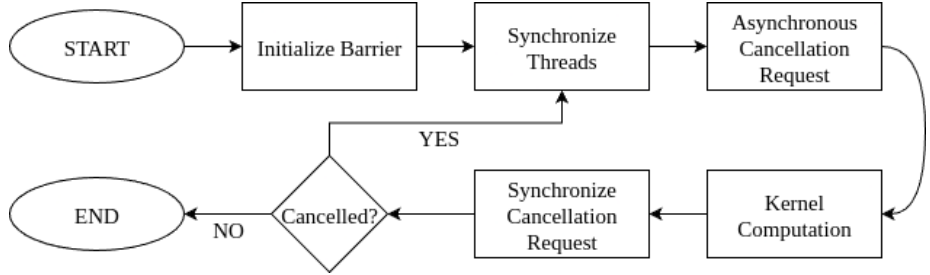

```
WG0: MMA A → epilogue A
WG1:          MMA B → epilogue B
```

- \+ epilogue / MMA overlap
- \+ reusable producer setup
- − more live state
- − harder termination
- − possible resource monopolization

CUTLASS, Hopper Warp Specialization — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/efficient_gemm.html
PyTorch Blog, CUTLASS Ping-Pong GEMM Kernel — https://pytorch.org/blog/cutlass-ping-pong-gemm-kernel/

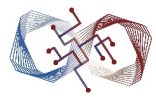
Cluster Launch Control: Combine Two Trade-offs

	low overhead	load balance	preemption opportunity
fixed work / block	—	✓	✓
fixed number of blocks	✓	—	—
CLC	✓	✓	✓



CUDA Programming Guide, Cluster Launch Control — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cluster-launch-control.html

A CLC Request Is an Async Producer

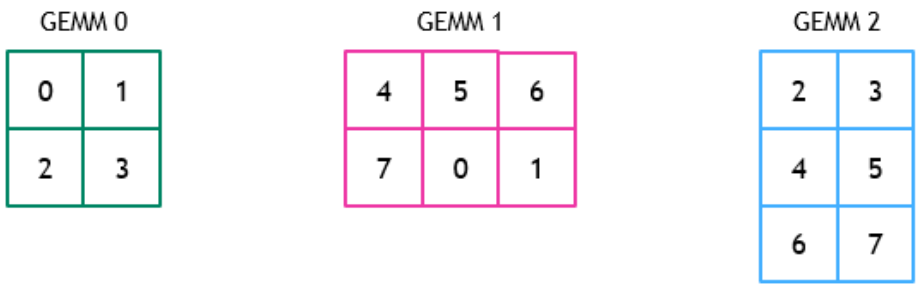


```
while (true) {
  issue_cancel_request(&result, &bar);  // elected thread
  compute(block_coordinate);

  wait_for_barrier_phase(&bar);
  if (!request_succeeded(result))
    break;

  block_coordinate = canceled_cta_id(result);
  release_result_for_next_request();
}
```

request
async proxy
→
mbarrier
completion
→
decode
next CTA id
→
proxy release
safe overwrite

CUDA Programming Guide, Cluster Launch Control API Details — https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cluster-launch-control.html

Grouped GEMM Creates One Logical Tile Space



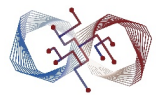
`[GEMM 0] [GEMM 1] [GEMM 2] ...`

- each rectangle = one output tile
- number = persistent block ID
- different groups may have different shapes
- one block computes several tiles

```
num_m[g]     = ceil(M_g / BLOCK_M)
num_n[g]     = ceil(N_g / BLOCK_N)
num_tiles[g] = num_m[g] × num_n[g]
```

CUTLASS, Grouped Kernel Schedulers — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/grouped_scheduler.html

Static Persistence: `tile_idx += NUM_SM`



```
grid = (NUM_SM, )
tile_idx = tl.program_id(0)

for g in range(group_size):
  ...
  while start ⩽ tile_idx < end:
    compute(g, tile_idx)
    tile_idx += NUM_SM
```

program 0

`0, 4, 8, 12`

program 1

`1, 5, 9, 13`

program 2

`2, 6, 10`

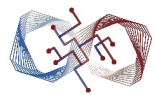
program 3

`3, 7, 11`

Fixed workers; a static round-robin schedule over global tile IDs.

Triton Tutorial, Group GEMM — https://triton-lang.org/main/getting-started/tutorials/08-grouped-gemm.html
CUTLASS, Grouped Kernel Schedulers — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/grouped_scheduler.html

A Global Tile ID Becomes `(g, tile_m, tile_n)`



```
num_m = cdiv(gm, BLOCK_M)
num_n = cdiv(gn, BLOCK_N)
num_tiles = num_m * num_n

if start ⩽ tile_idx < start + num_tiles:
  local  = tile_idx - start
  tile_m = local // num_n
  tile_n = local %  num_n

start += num_tiles
```

group ranges

`g0 [0,6) · g1 [6,8) · g2 [8,14)`

global ID

`tile_idx = 9`

group + local ID

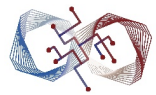
`g = 2 · local = 9 - 8 = 1`

tile coordinates

`num_n = 2 → (tile_m,tile_n) = (0,1)`

Triton Tutorial, Group GEMM — https://triton-lang.org/main/getting-started/tutorials/08-grouped-gemm.html
CUTLASS, Grouped Kernel Schedulers — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/grouped_scheduler.html

At a Group Boundary, Bindings Change

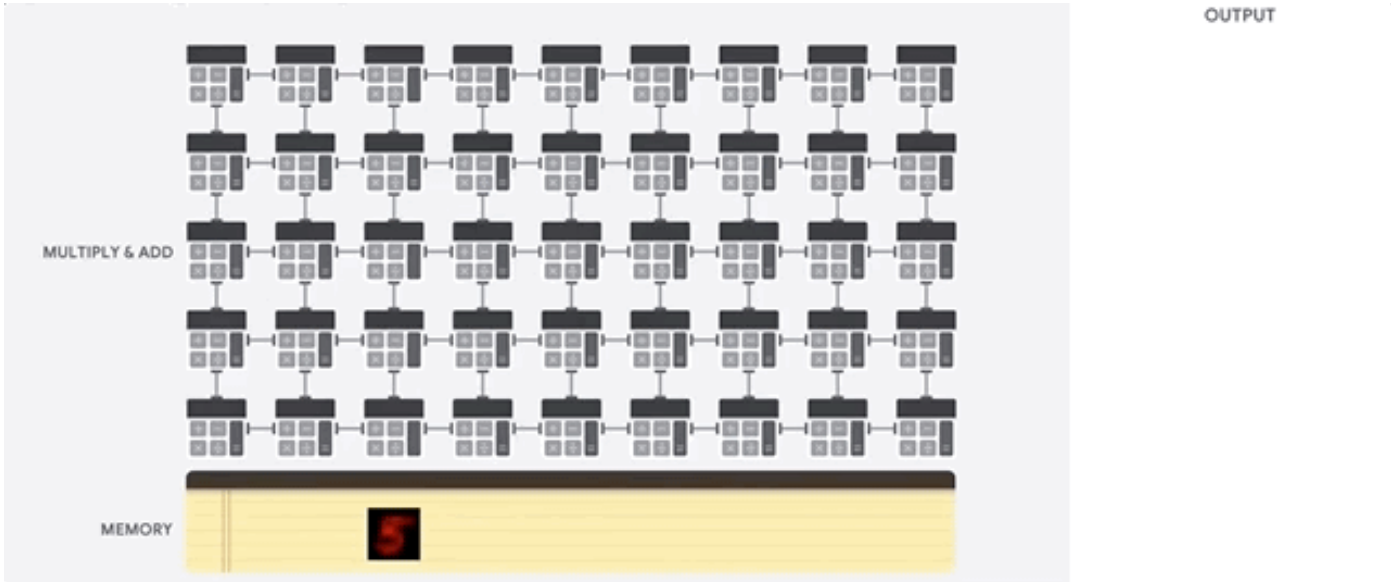


roles + stage protocol	→	persist
group ID	→	changes
A/B/C/D + strides	→	rebind
descriptors + predicates	→	update
epilogue destination	→	D[group]

The pipeline survives the boundary; its data bindings do not.

CUTLASS, Grouped Kernel Schedulers — https://docs.nvidia.com/cutlass/latest/media/docs/cpp/grouped_scheduler.html
NVIDIA CUTLASS Example 57 — https://github.com/NVIDIA/cutlass/blob/main/examples/57_hopper_grouped_gemm/57_hopper_grouped_gemm.cu

TPU Before Multiply: Parameters Move HBM → MXU



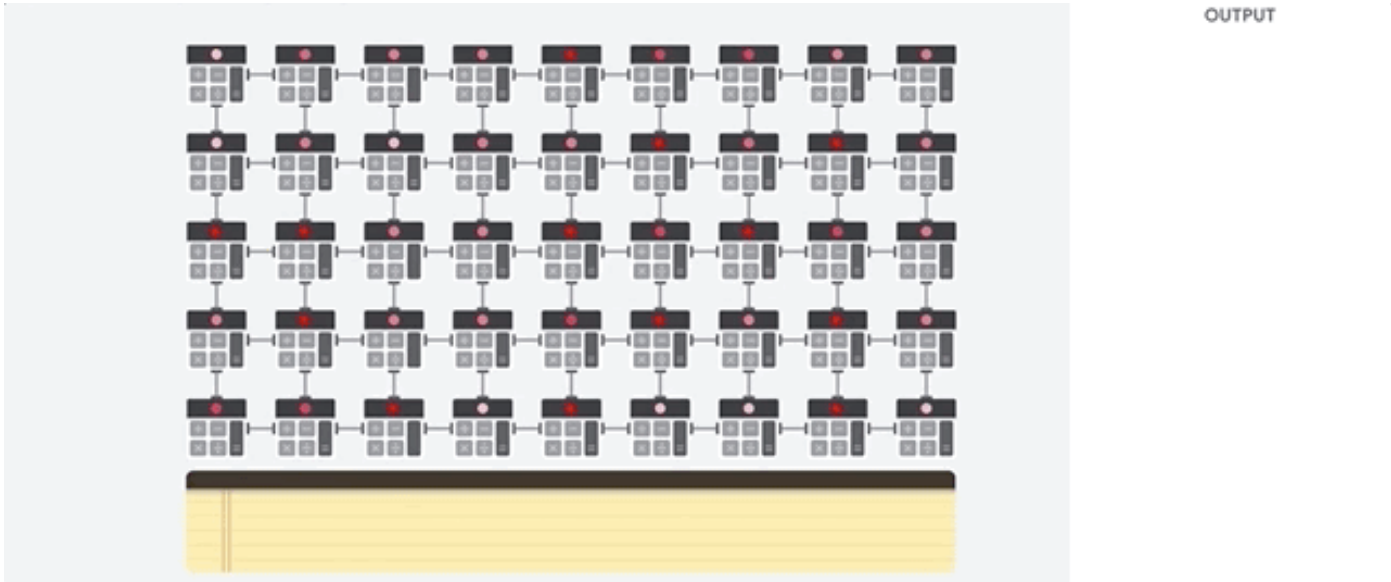
HBM
holds data and parameters

parameters
move toward the compute array

MXU
receives them before multiplication

Google Cloud, TPU Architecture — https://docs.cloud.google.com/tpu/docs/system-architecture-tpu-vm

Inside the MXU: Data Moves Between MACs



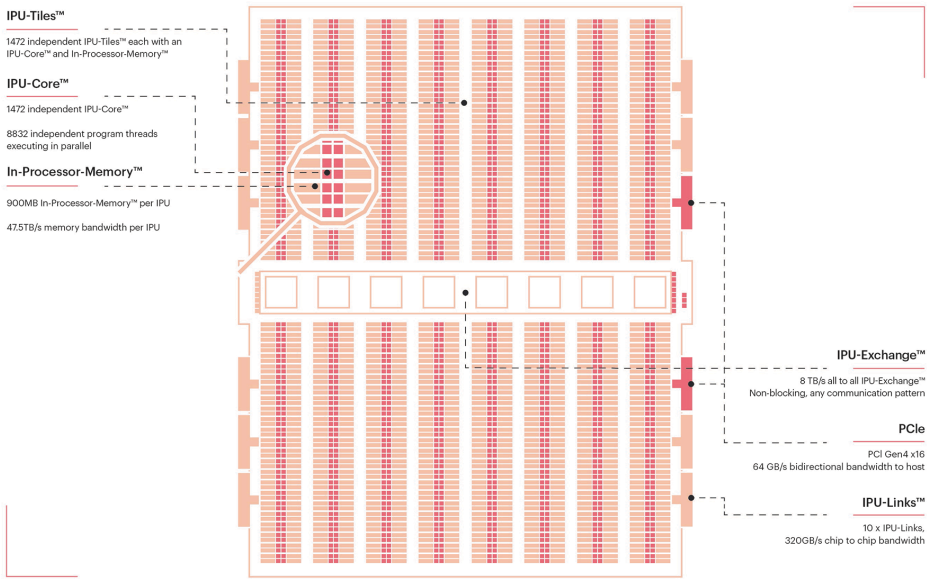
inject
data enters the array edge

multiply + forward
each MAC passes data onward

collect
results return toward memory

Google Cloud, TPU Architecture — https://docs.cloud.google.com/tpu/docs/system-architecture-tpu-vm

IPU: Compute Sits Beside Local Memory



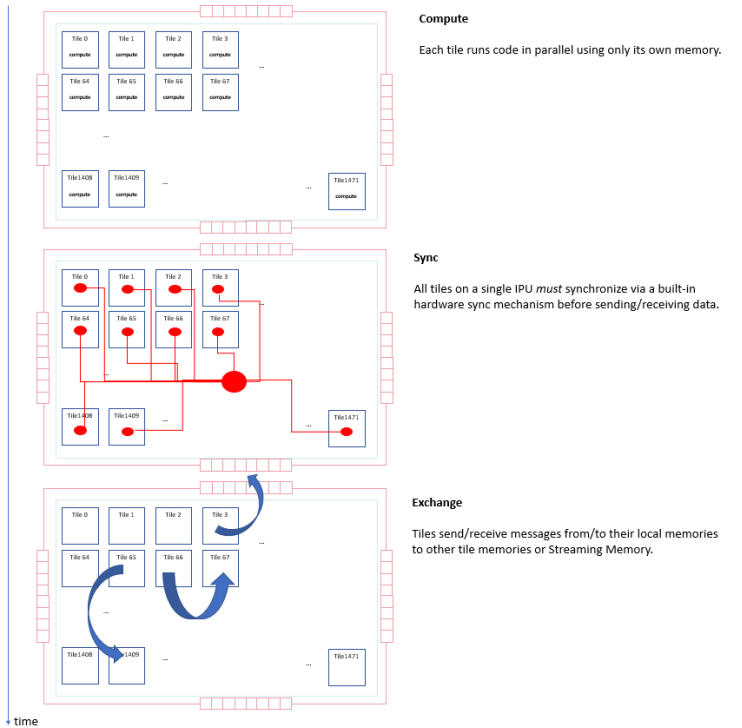
IPU tile
core + local in-processor memory

Exchange
moves non-local data between tiles

Compiler
places code, data, and communication

Graphcore, IPU Programmer's Guide — https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about_ipu.html
Jia et al., Dissecting the Graphcore IPU Architecture via Microbenchmarking — https://arxiv.org/abs/1912.03413

IPU Execution: Compute → Sync → Exchange



Compute

local tile memory

Sync

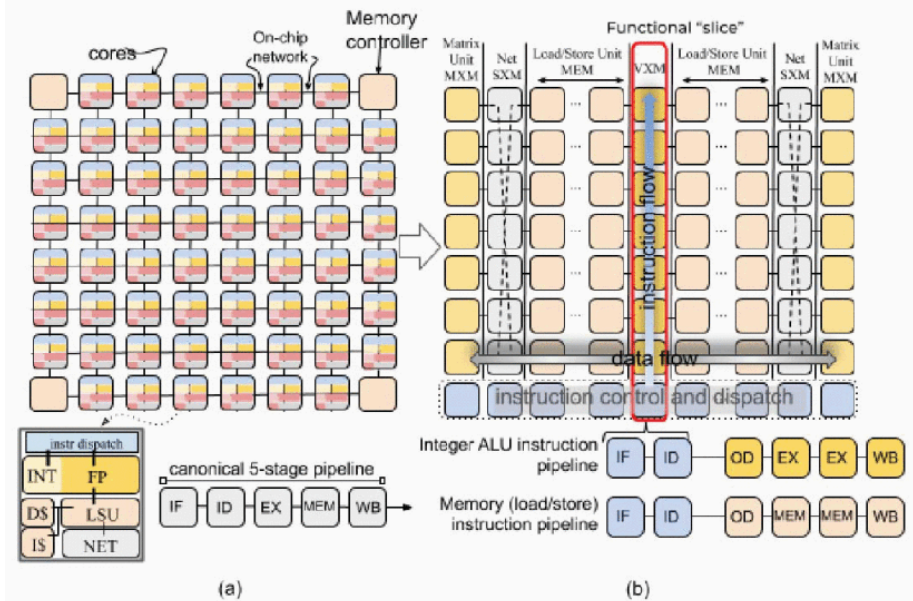
all tiles reach the phase boundary

Exchange

explicit movement over the fabric

Graphcore IPU Programmer's Guide, Execution Model — https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about_ipu.html

TSP Reorganizes Cores into Functional Slices



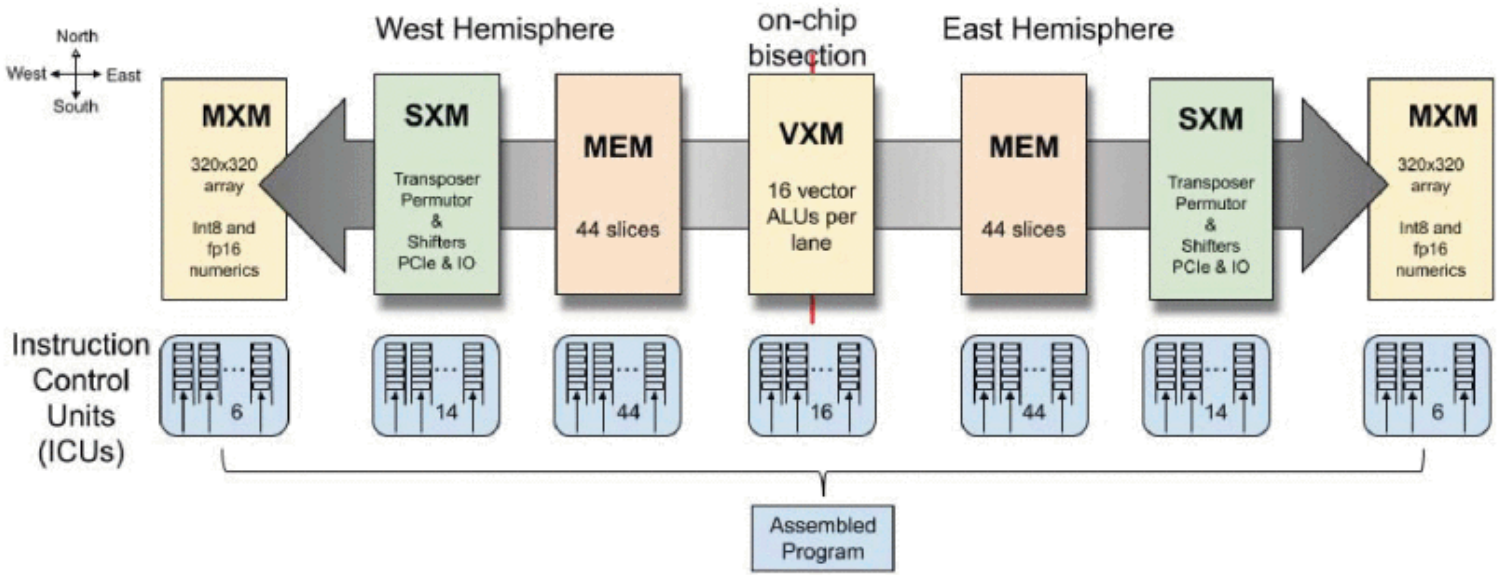
conventional core
many functions inside every tile

functional slice
one function repeated vertically

two directions
instructions ↑ · data ↔

Abts et al., Think Fast: A Tensor Streaming Processor, ISCA 2020 — https://ieeexplore.ieee.org/document/9138986

TSP Connects Functional Slices with Streams



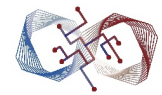
MEM
reads and writes stream operands

SXM · VXM · MXM
route, vector, and matrix functions

compiler
aligns instruction and operand arrival

Abts et al., Think Fast: A Tensor Streaming Processor, ISCA 2020 — https://ieeexplore.ieee.org/document/9138986

Where Does Uncertainty Live?



model	main orchestration	principal trade-off
GPU pipeline	dynamic readiness + explicit waits	buffering / occupancy
TPU systolic array	spatial wave through MAC cells	shape rigidity / fill-drain
Graphcore IPU	placement + BSP exchange phases	mapping / global imbalance
Groq TSP	functional slices + static stream schedule	compiler burden / less dynamism