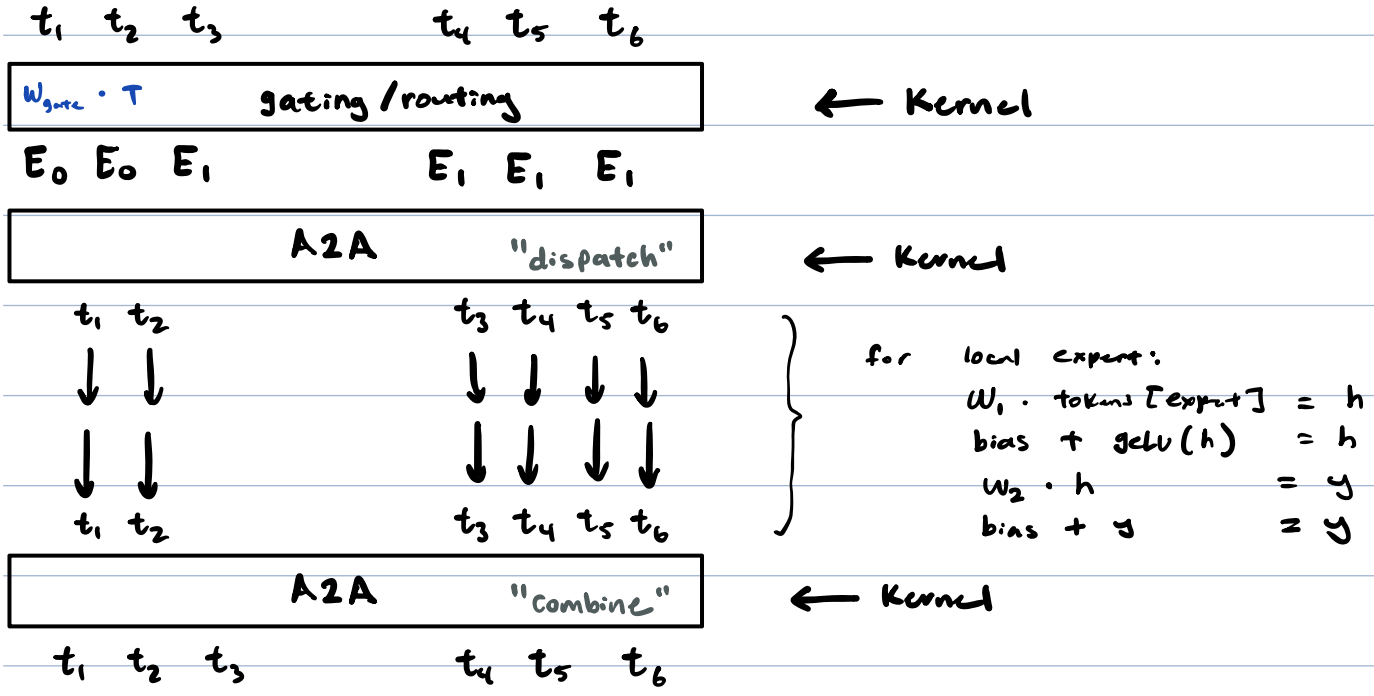
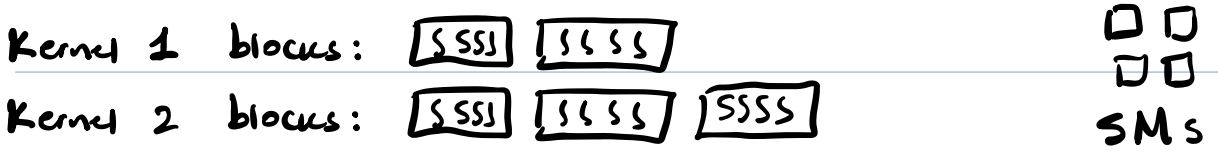
Flash MoE

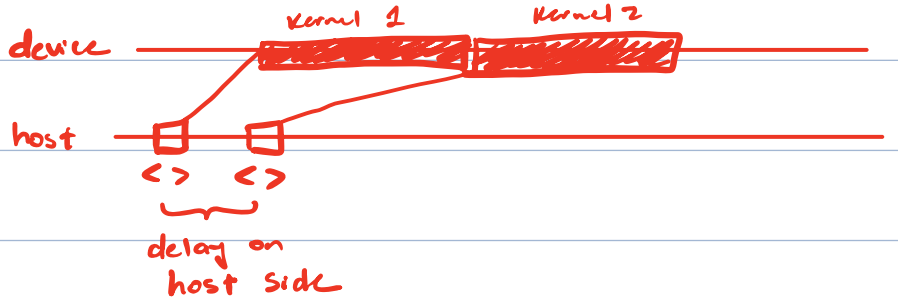
Traditional MoE:



Traditional Kernel



Problem 1: host must launch the kernel

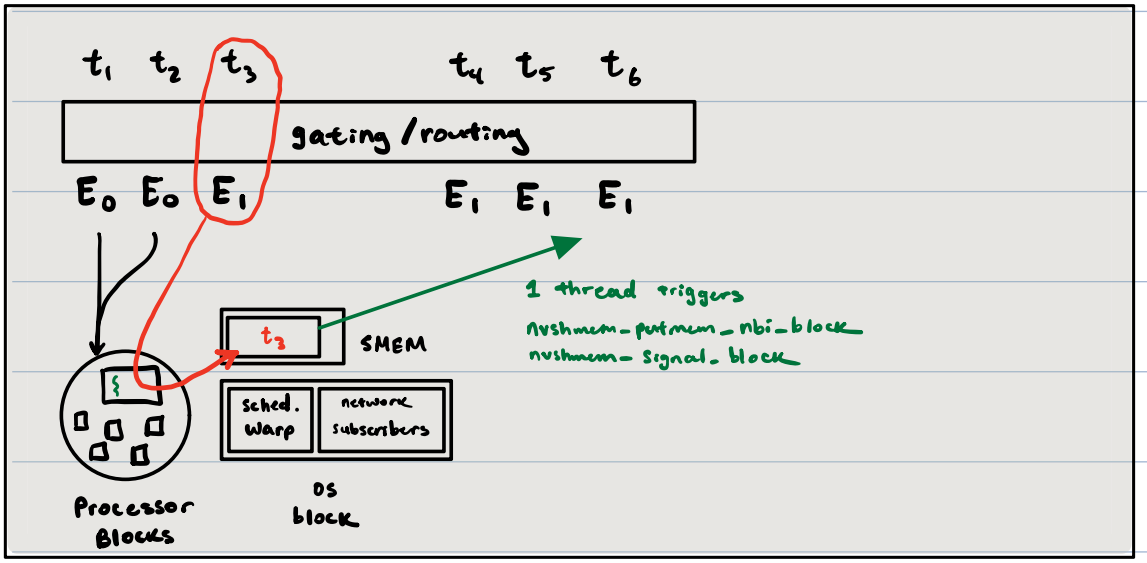


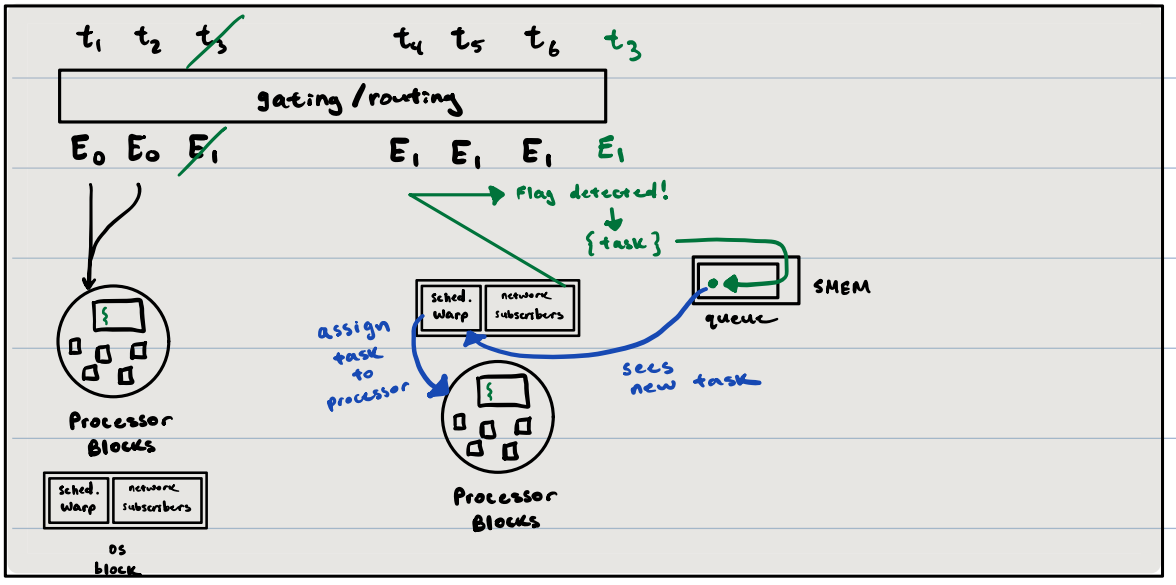
Problem 2: ALL blocks on K_1 must finish before ANY block on K_2 can run!

Flash D MoE

```
flashMoE <<< N , 128 >>> ( params )
```

- N: # blocks = 2 × # SMs (2 SMs/block lets you interleave & use ALL registers on the SM)
- 128: threads / block = 4 quadrants × 32 threads / quadrant






```
Task_descriptor = {
    ptr input tile
    ptr expert weights
    ptr output
    tileId
    PeerId
    expertId
    taskType (GEMM0/1, Combine)
    <- Padding ->
}
```

The One True Kernel

input seq. · output · experts

```
__global__ void flashMoE(A /*[S,H]*/, O /*[S,H]*/, X /*[E,H,D] expert weights*/, N)
{
    // ---- Phase 1: every block cooperates on the gate ----
    // T_phi[e,c] = (token_id, combine_weight);  G_phi[s,e] = affinity scores
    (T_phi, G_phi) = FusedGate(A);

    if (blockId < N - 1) {
        // ---- Phase 2a: compute blocks ----
        Dispatch(T_phi, A);     // pack my tokens into packets, nvshmem_put + signal
                                // to every peer that owns a selected expert
        processor::start();     // never returns until interrupted
    } else {
        // ---- Phase 2b: the OS block ----
        if (warpId == 0) scheduler::start();
        else             subscriber::start(T_phi, G_phi, O, X);
    }
}
```

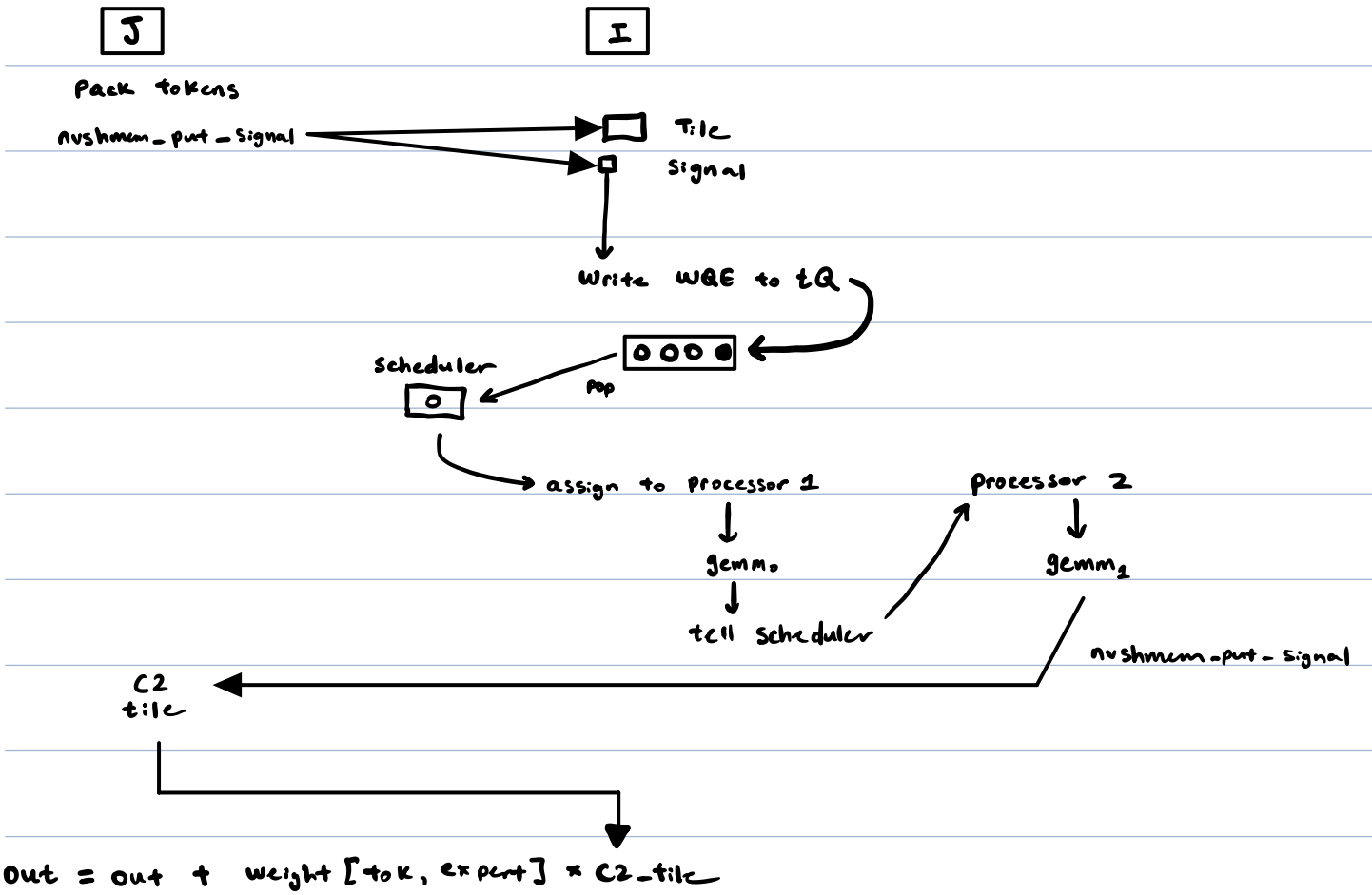
ALL blocks help compute routing gate

N Processors

W0: Scheduler
Wx: Subscriber

last block!

Life of a tile



```
void processor::start() {
    tQ = GetTQ();                       // global task queue
    __shared__ Task task;               // shared: one fetch, all 128 threads read
    __shared__ bool interrupt, complete;

    while (!interrupt) {
        if (warpId == 0) {
            if (threadId == 0) {
                awaitTaskFromScheduler(&interrupt, &signal);  // spin on my doorbell
                FencedNotifyRQ(ready);                        // "I'm available"
            }
            __syncwarp();
            warpReadTQ(tQ, signal, &task);   // 32 threads cooperatively load
        }                                    // the 128B task descriptor
        __syncthreads();                     // whole block now sees the task

        if (interrupt) break;

        switch (task.taskType) {

        case GEMM0:
            fGET(GEMM0, task);   // fused GEMM + epilogue(GELU) + async staging
            if (threadId == 0)
                complete = NotifyTileCompletion();   // atomic on row-block counter
            __syncthreads();
            if (complete) NotifySchedulerNextGEMM(tQ);   // release GEMM1 tasks
            break;

        case GEMM1:
            fGET(GEMM1, task);   // fused GEMM + epilogue + async tile TRANSFER
            break;               // the nvshmem_put is *inside* the epilogue

        case Combine:
            combine(task);       // C ← A ⊙ S + C  (scale by affinity, accumulate)
            break;
        }
    }
}
```

Smem vars:

- Task Queue
- Task
- Interrupt
- Complete

1 thread spins looking for tasks

Warp-load into Task

READ TASK TYPE

gemm,
§ thread signal completion

schedule next gemm

gemm, then nvshmem_put

Finished gemm,
so accumulate output

```
void scheduler::start() {      // ONE warp, 32 threads
    scheduled = 0;
    tTB = atomicLoad(taskBound);        // upper bound, corrected at runtime
    readyQueue = circular_buffer(all processor ids);

    while (scheduled < tTB) {
        // 32 threads sweep 32 doorbells in parallel
        localTasks = 0;
        parallel_for (doorbell in myDoorbells)
            localTasks += observe(doorbell);

        // prefix sum gives each lane a disjoint slice of the ready queue
        warpInclusiveSum(localTasks, &qS, &taskTally);

        while (taskTally > 0) {
            refill(readyQueue);          // harvest processors that signalled ready
            parallel_for (lane)
                signal(readyQueue[qS + lane], taskIndex);
        }

        if (threadId == 0) tTB = atomicLoad(taskBound);
        tTB = warpBroadcast(tTB);
    }
    InterruptSubscribers();
    InterruptProcessors();
}
```

While we still have tasks:

Sweep doorbells:
32 lanes each handle
1 doorbells
for new tasks

↑ accumulate # new tasks

add up all doorbells in the warp w/ warp-shfl--

Signal & hand out tasks to idle processors!

network subscriber:

```
void subscriber::start(T_phi, G_phi, O, X) {   // THREE warps = 96 threads
    while (!atomicLoad(interrupt)) {

        parallel_for (flag in dispatchFlags) {      // 96 flags polled at once
            sig = atomicLoad(flag);
            if (isSet(sig) && !visited(flag)) {
                markVisited(flag);
                SelfCorrectTaskBound(taskBound, sig);   // <-- see §9
                __threadfence_system();                 // consistency before read
                tasks = decodePacketToGEMM0Tasks(sig, X);
                writeTo(tQ, tasks);
                notifyScheduler(tasks.count);           // shared-mem doorbell
            }
        }

        parallel_for (flag in combineFlags) {       // one per returning tile
            sig = atomicLoad(flag);
            if (isSet(sig) && !visited(flag)) {
                markVisited(flag);
                __threadfence_system();
                tasks = decodeToCombineTasks(sig, T_phi, G_phi, O);
                writeTo(tQ, tasks);
                notifyScheduler(tasks.count);
            }
        }
    }
}
```

poll dispatch tasks

decode packet → WQE
enqueue
notify scheduler

poll combine tasks

decode packet → WQE
enqueue
notify scheduler

★ Symmetric Memory is organized such that it's **LOCK-FREE**: buffers & experts organized correctly!