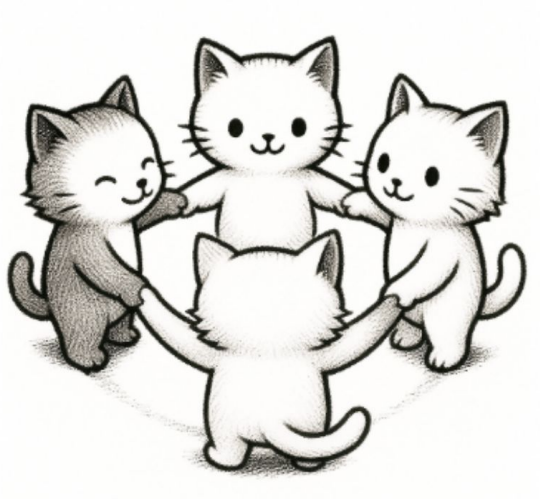
# ParallelKittens: Systematic and Practical Simplification of Multi-GPU AI Kernels

Stuart H. Sul, Simran Arora, Benjamin F. Spector, Christopher Ré

MLSYS 2026

# Preview

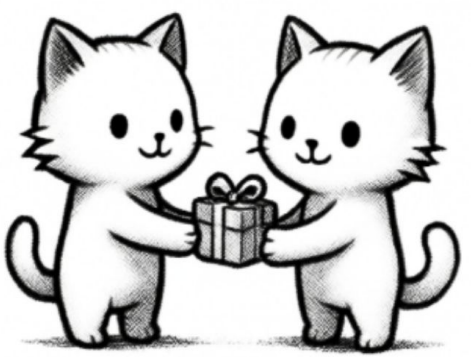
1. Motivation
    - Why GPU networking now?
    - 50% / HW advancements
2. Problem
    - Overlapping
    - Full exploitation
    - Simplicity



# Preview

3. Three trade-offs

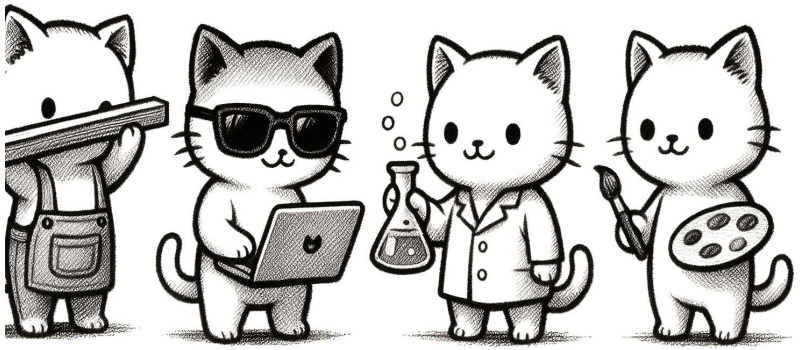
- Transfer mechanisms (7x)
- Scheduling (4x)
- Design overheads (1.8x)



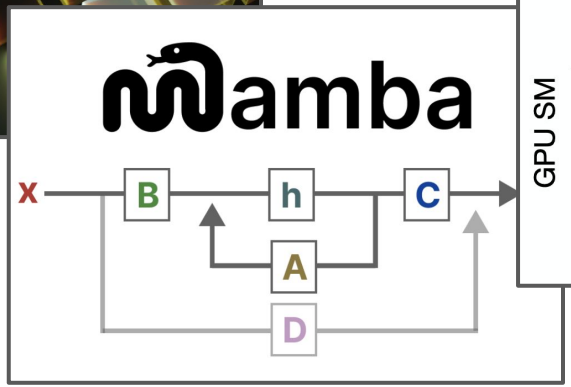
4. ParallelKittens: Multi-GPU kernel DSL

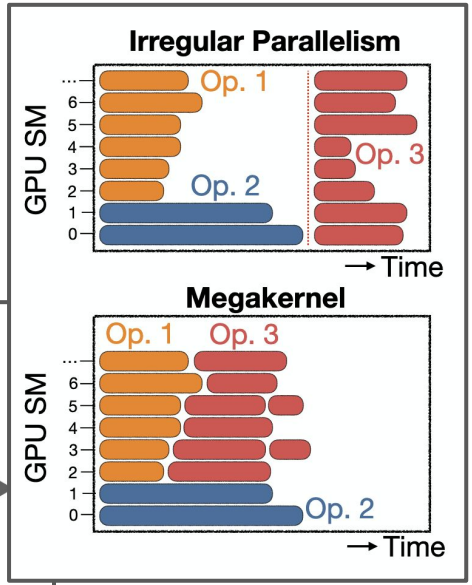
5. Results: reducing networking overhead to <1% / production usage

# Motivation



# Networking is the Remaining Bottleneck

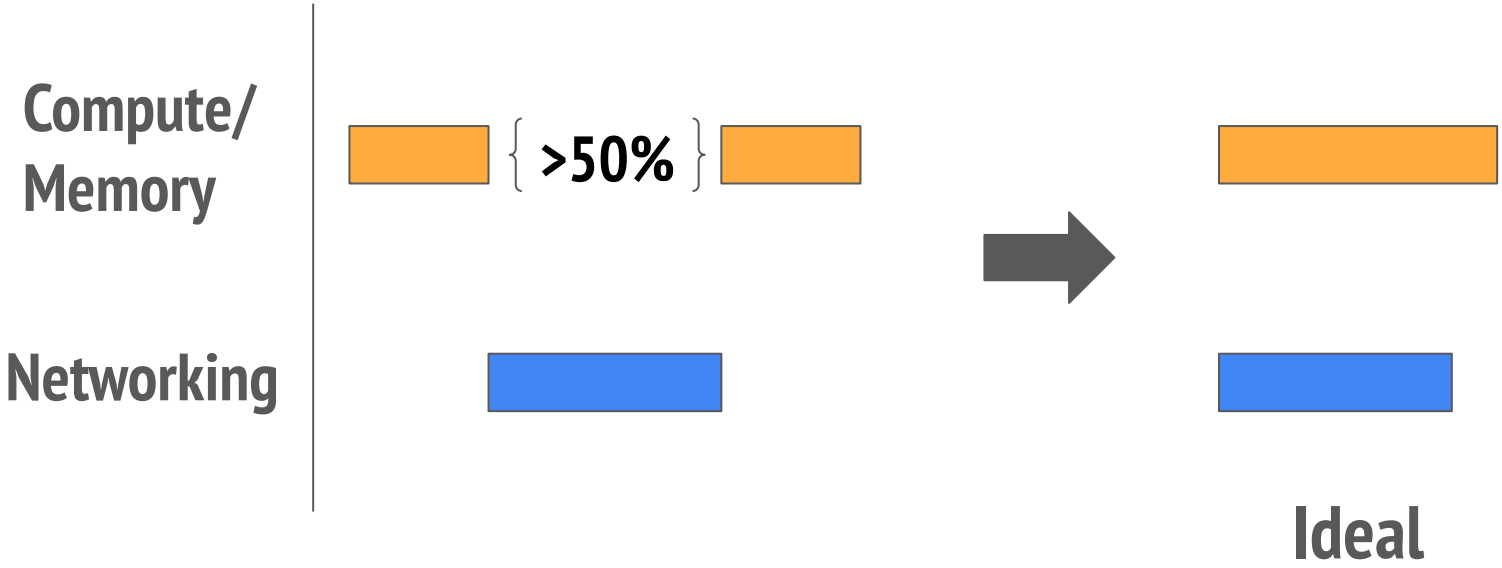



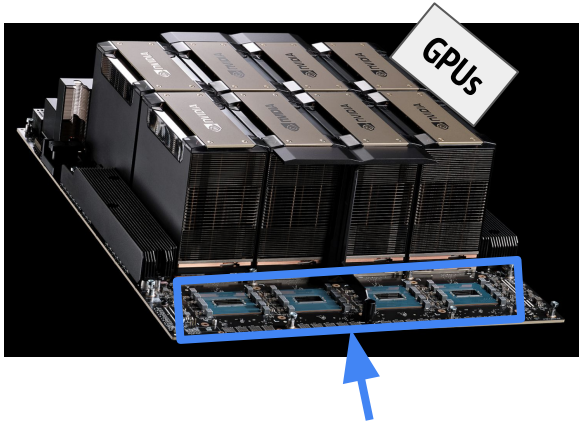

CuTe DSL

Mojo🔥

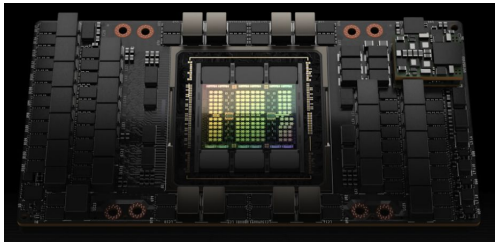
# Networking Wastes GPU Utilization



# HW Advancements Open New Opportunities




In-network Compute



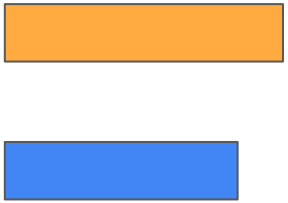
Async & Bulk Device-initiated Networking

Scale-up Arch

# Problem

# Challenge:

## It is difficult to write multi-GPU kernels with good:



1. Overlap

2. HW Exploitation


3. Simplicity

# Practitioners Have Three Options:

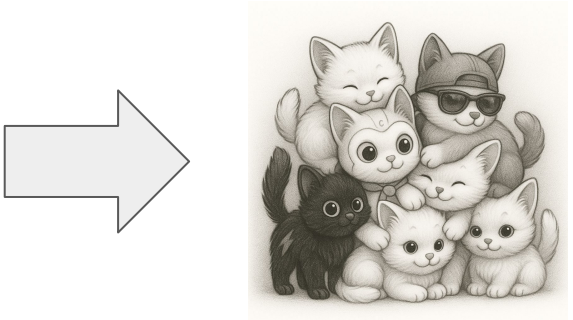
Off-the-shelf Libraries

Compiler-based Approaches

Low-level Primitives

# Question:

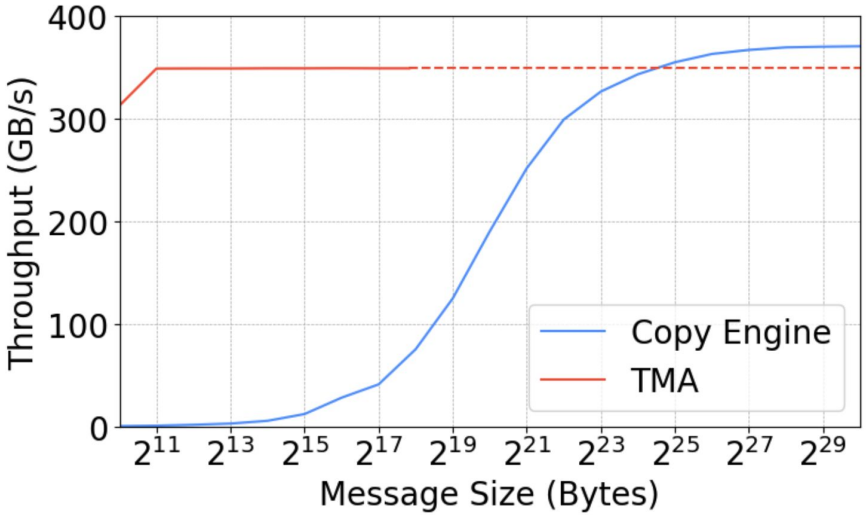
Is there a small set of principles that clarifies the hardware tradeoffs and multi-GPU AI kernel design space, leading to a minimal set of programming abstractions / primitives?



# The Three Trade-offs

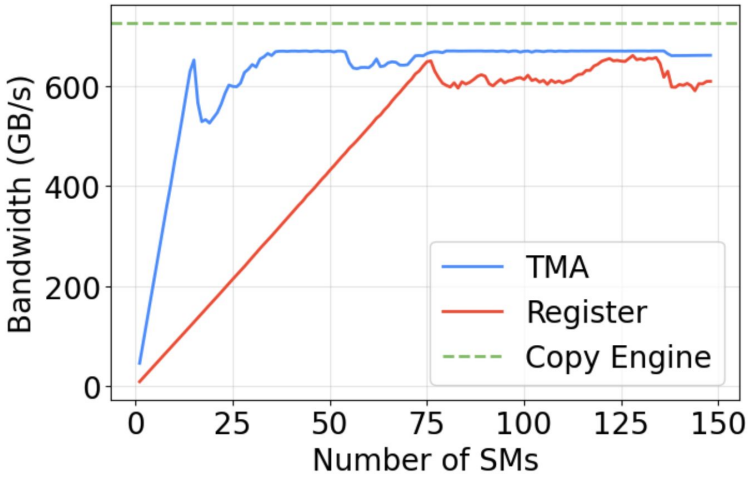
# Trade-off #1: Transfer Mechanism

1. Copy Engine
2. TMA
3. Register Instructions



# Trade-off #1: Transfer Mechanism

1. Copy Engine
2. TMA
3. Register Instructions



# Trade-off #1: Transfer Mechanism

1. Copy Engine
2. TMA
3. Register Instructions

| Functionality | CE | TMA | Reg |
|---|---|---|---|
| P2P Transfer | ✓ | ✓ | ✓ |
| In-fabric Broadcast | ✓ | ✓ | ✓ |
| P2P Reduction | × | ✓ | ✓ |
| In-fabric Reduction | × | × | ✓ |
| Elementwise Transfer | × | × | ✓ |

# Trade-off #1: Transfer Mechanism

| **AG-GEMM** | **Triton Distributed** | **Flux** | **CUTLASS** | **PK** |
|---|---|---|---|---|
| **Transfer Mechanism** | Copy Engine | Copy Engine | Copy Engine | TMA |

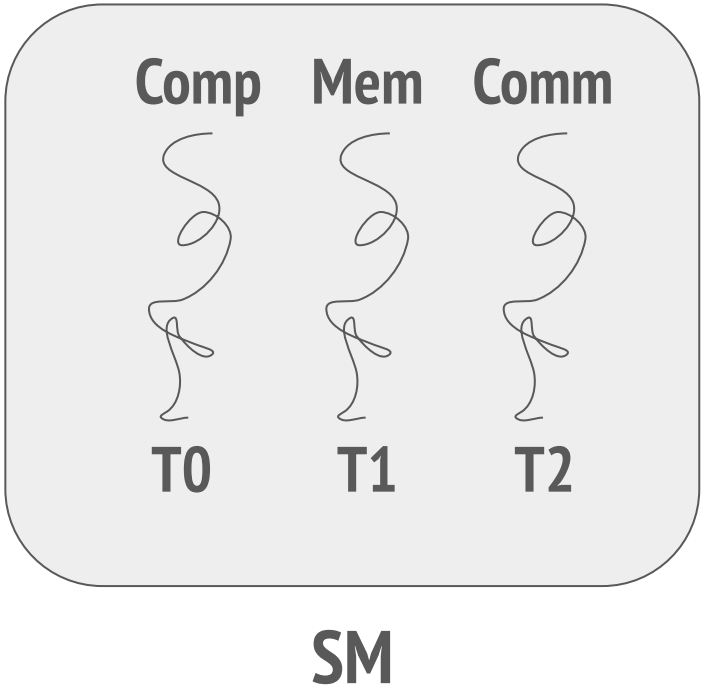
**0.97x–7.39x (fine-grained)**

# Trade-off #2: Scheduling

1. Intra-SM Overlapping
2. Inter-SM Overlapping

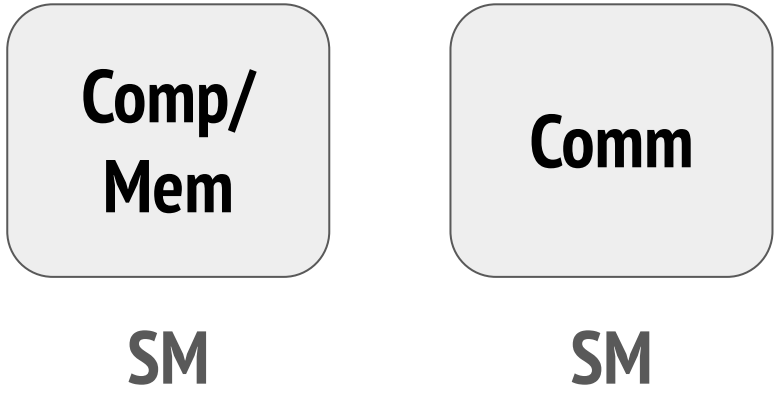
# Trade-off #2: Scheduling

1. Intra-SM Overlapping
2. Inter-SM Overlapping



# Trade-off #2: Scheduling

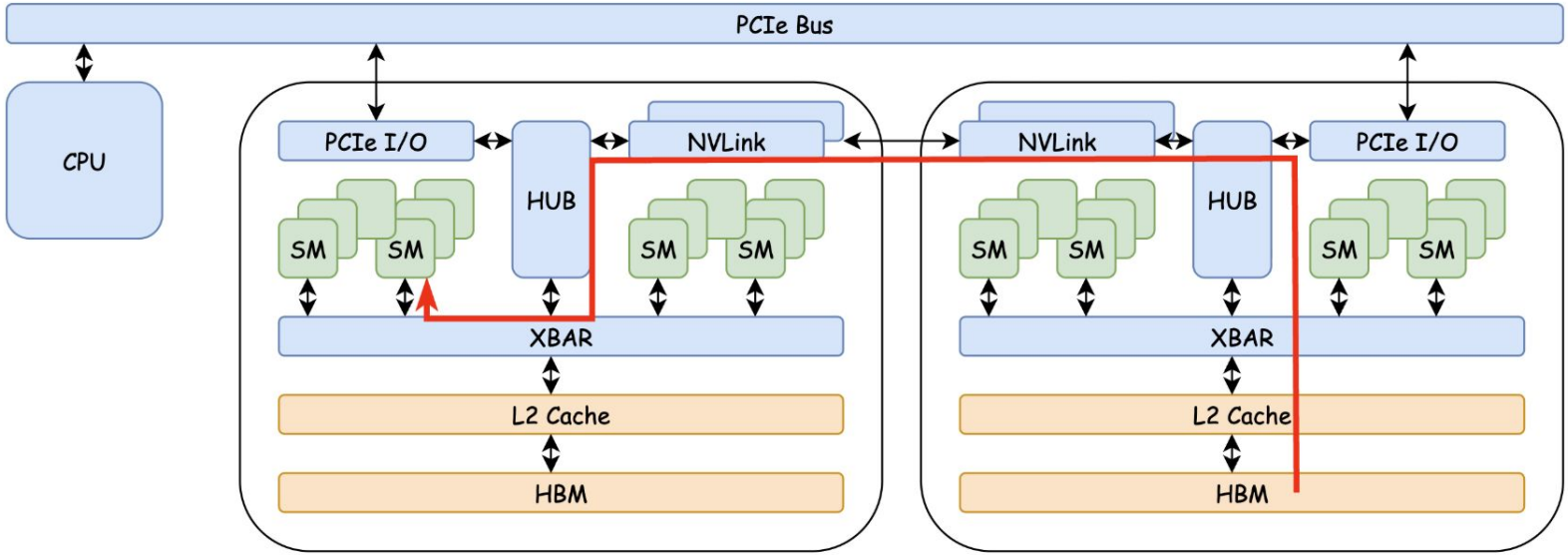
1. Intra-SM Overlapping
2. Inter-SM Overlapping



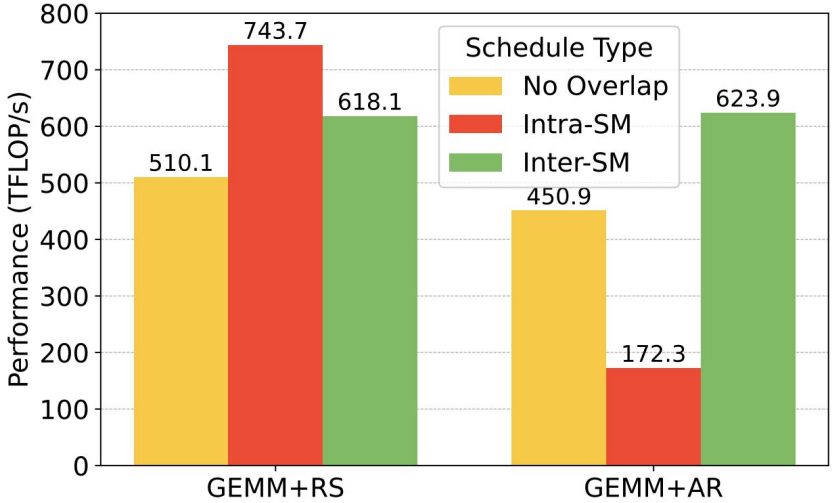
# Trade-off #2: Scheduling

| | Intra-SM | Inter-SM |
|---|---|---|
| Full SM utilization | ✅ | ❌ |
| No inter-SM sync | ✅ | ❌ |
| Tolerates comp/comm misalignment | ❌ | ✅ |
| Efficient in-network compute & local prefetching | ❌ | ✅ |

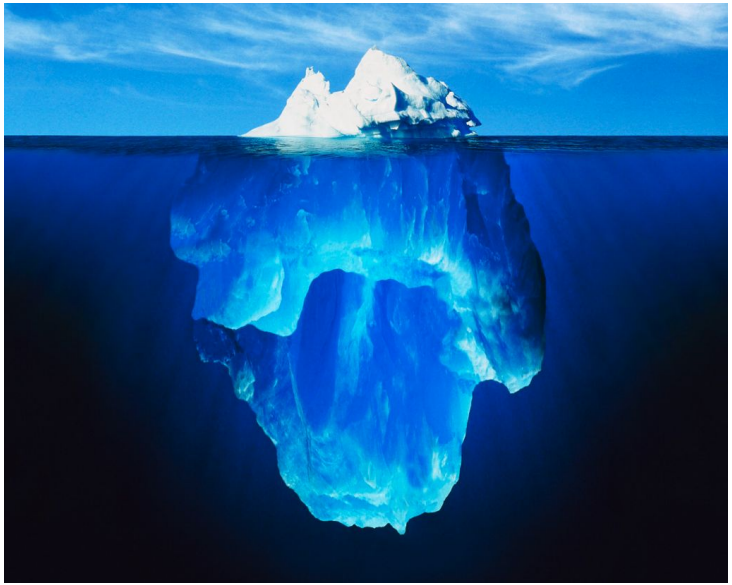
# Trade-off #2: Scheduling



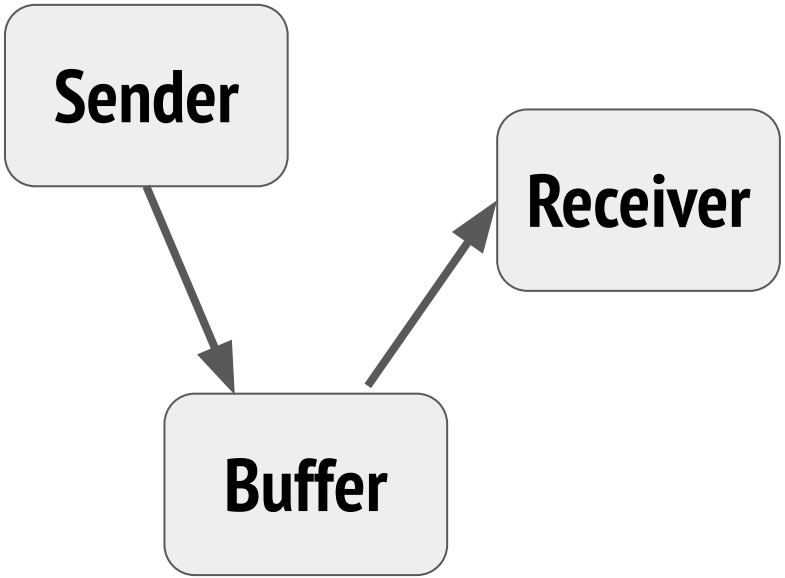
# Trade-off #2: Scheduling

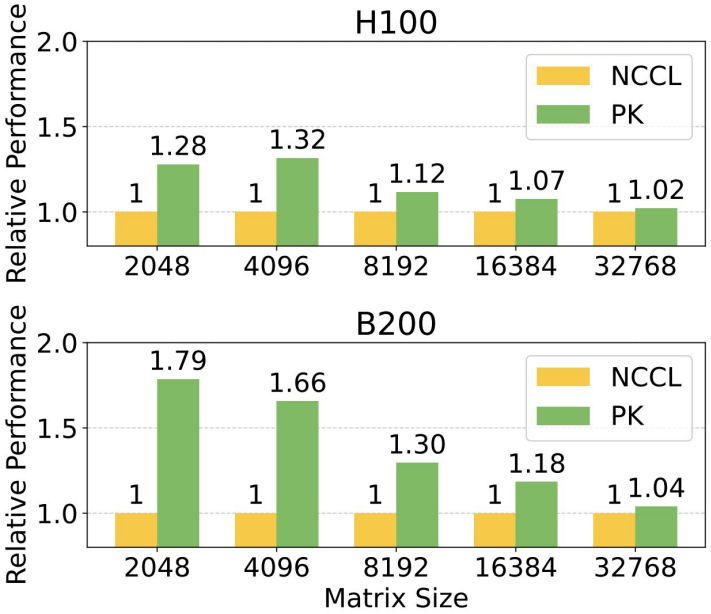


# Trade-off #3: Design Overheads

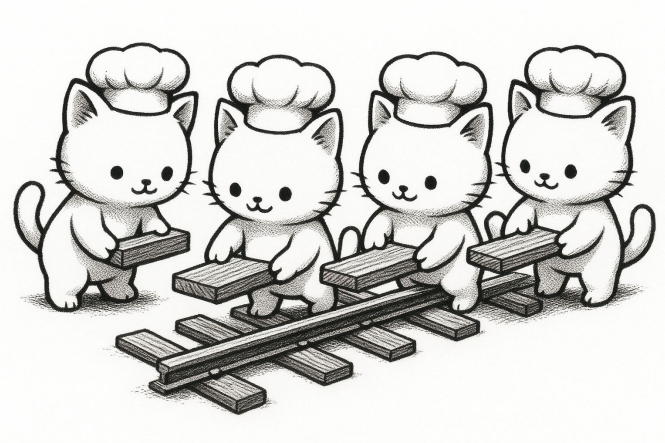


# Trade-off #3: Design Overheads



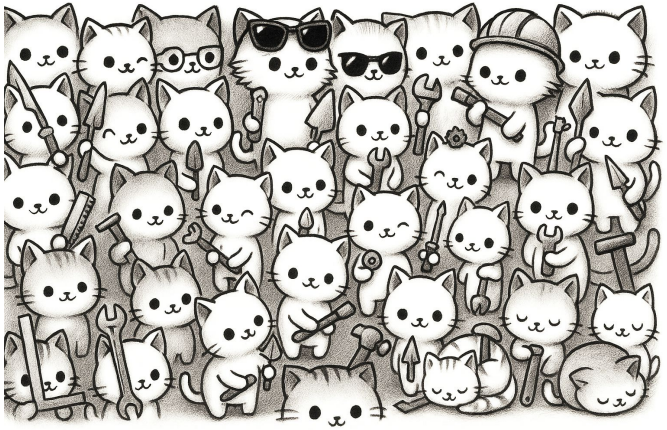



# ParallelKittens

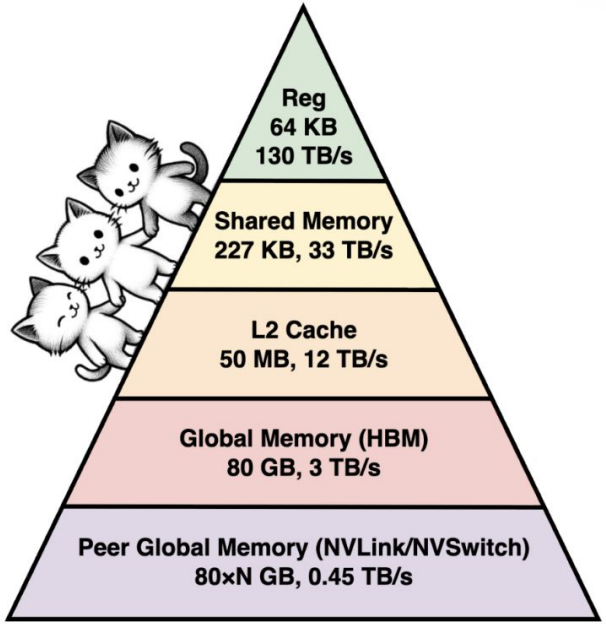


# ParallelKittens

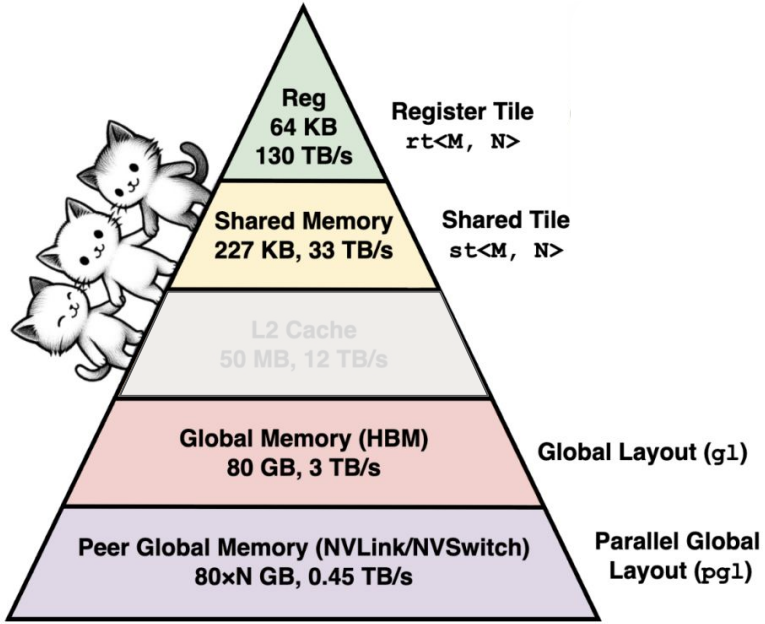
## Highly opinionated set of CUDA programming primitives that extends ThunderKittens



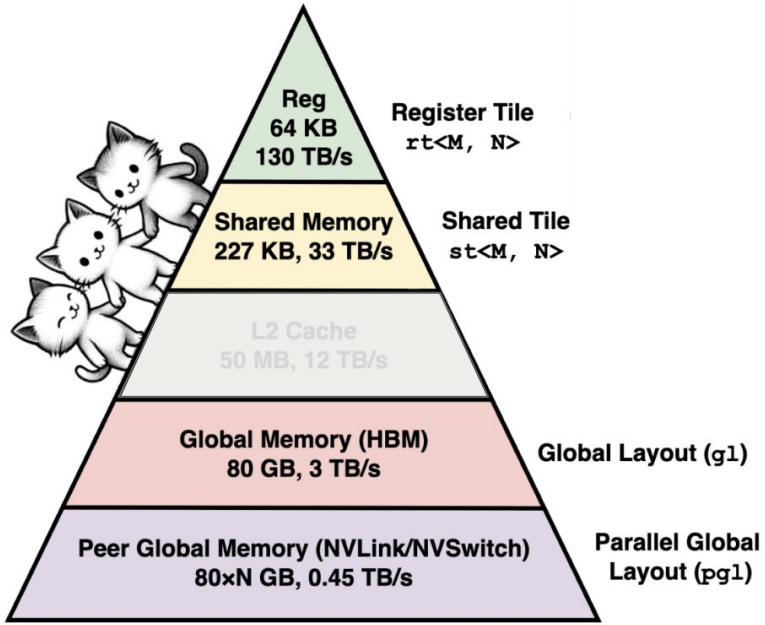
# ParallelKittens



# ParallelKittens



# ParallelKittens




## P2P communication

- `store_async(dst, src, coord)`
- `store_add_async(dst, src, coord)`

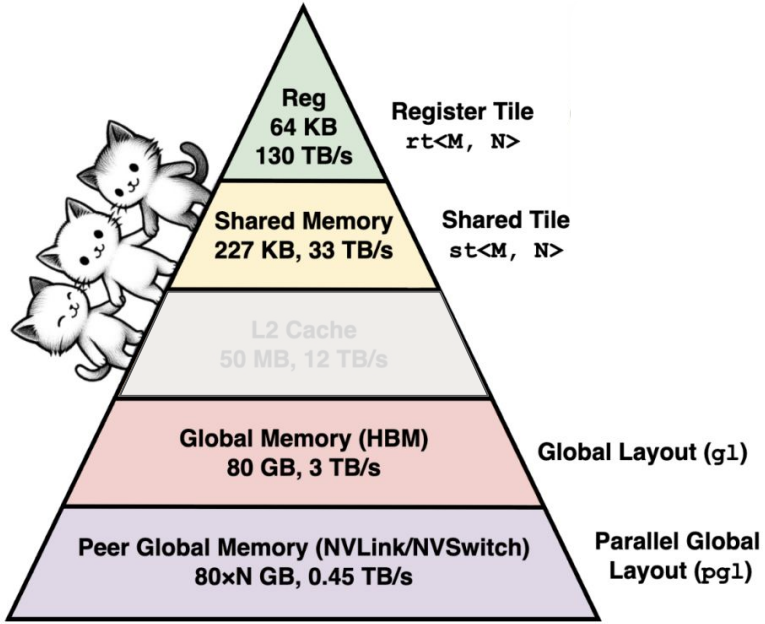
## Network-accelerated communication

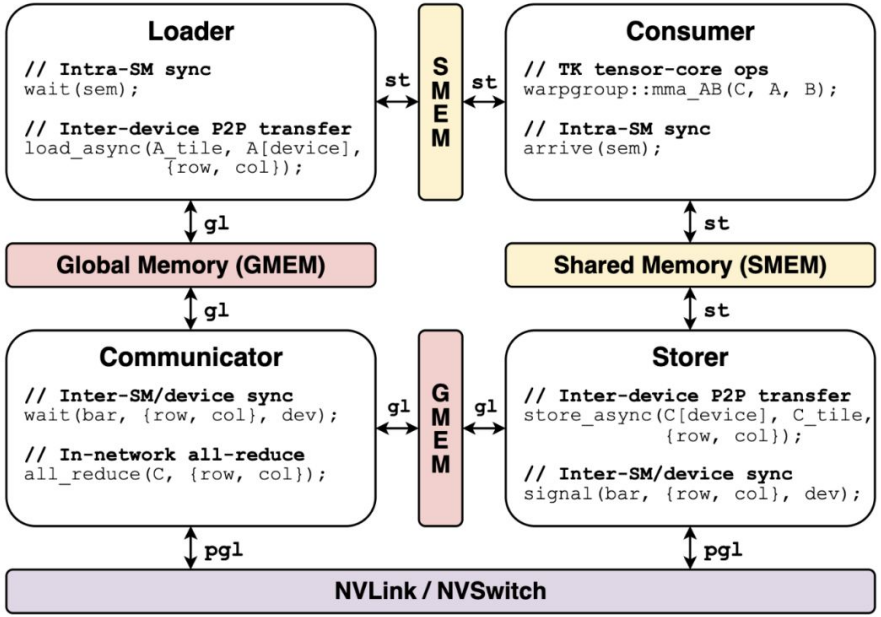
- `reduce(dst, dst coord, src, src coord)`
- `all reduce(dst and src, coord)`

## Inter-device/SM synchronization

- `signal(bar, coord, dev idx, val)`
- `signal all(bar, coord, val)`
- `wait(bar, coord, dev idx, expected)`
- `barrier(bar, coord, dev idx)`

# ParallelKittens



# ParallelKittens: Example Code

```
__device__ inline void loader(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    for (int red_idx = 0; red_idx < regs.num_iters; red_idx++) {
        wait(sem.inputs_finished[regs.stage], get_phasebit<1>(regs.phasebits, regs.stage));
        update_phasebit<1>(regs.phasebits, regs.stage);
        tma::expect_bytes(sem.inputs_arrived[regs.stage], sizeof(A_tile) * 2 + sizeof(B_tile));
        if (red_idx == PIPELINE_STAGES - 1) {
            wait(sem.outputs_finished, get_phasebit<1>(regs.phasebits, PIPELINE_STAGES));
            update_phasebit<1>(regs.phasebits, PIPELINE_STAGES);
        }
        for (int i = 0; i < 2; i++)
            tma::load_async(smem.inputs[regs.stage].A[i], G.A, {idx.x * 2 + i, red_idx}, sem.
                ↪ inputs_arrived[regs.stage]);
        tma::load_async(smem.inputs[regs.stage].B, G.B, {red_idx, idx.y}, sem.inputs_arrived[regs.stage])
            ↪ ;
        regs.stage = (regs.stage + 1) % PIPELINE_STAGES;
    }
}

__device__ inline void storer(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    wait(sem.outputs_arrived, get_phasebit<0>(regs.phasebits, 0));
    update_phasebit<0>(regs.phasebits, 0);
    for (int i = 0; i < 2; i++)
        tma::store_async(G.C[G.dev_idx], regs.C[i], {idx.x * 2 + i, idx.y});
    tma::store_async_read_wait();
    arrive(sem.outputs_finished);
    int signal_dev_idx = regs.task_id % NUM_DEVICES;
    device<NUM_DEVICES>::signal(G.barrier, {idx.x, idx.y}, signal_dev_idx, 1);
}

__device__ inline void consumer(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    rt_fl<ROW_BLOCK / 8, COL_BLOCK> C_accum;
    warp::zero(C_accum);
    for (int red_idx = 0; red_idx < regs.num_iters; red_idx++) {
        wait(sem.inputs_arrived[regs.stage], get_phasebit<0>(regs.phasebits, regs.stage));
        update_phasebit<0>(regs.phasebits, regs.stage);
        warpgroup::mma_AB(C_accum, smem.inputs[regs.stage].A[regs.warpgroup_id], smem.inputs[regs.stage].
            ↪ B);
        warpgroup::mma_async_wait();
        warp::arrive(sem.inputs_finished[regs.stage]);
        regs.stage = (regs.stage + 1) % PIPELINE_STAGES;
    }
    group<8>::sync(3);
    warpgroup::store(regs.C[regs.warpgroup_id], C_accum);
    warpgroup::sync(regs.warpgroup_id + 1);
    warpgroup::arrive(sem.outputs_arrived);
}

__device__ inline void communicator(const globals &G, comm_sem &sem, comm_smem &smem, comm_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    if (threadIdx.x == 0)
        device<NUM_DEVICES>::wait(G.barrier, {idx.x, idx.y}, G.dev_idx, NUM_DEVICES);
    __syncthreads();
    group<NUM_WARPS>::all_reduce<ROW_BLOCK, COL_BLOCK, reduce_op::ADD>(G.C, {idx.x, idx.y});
}
```

Loader

Storer

Consumer

Communicator

# ParallelKittens: Example Code

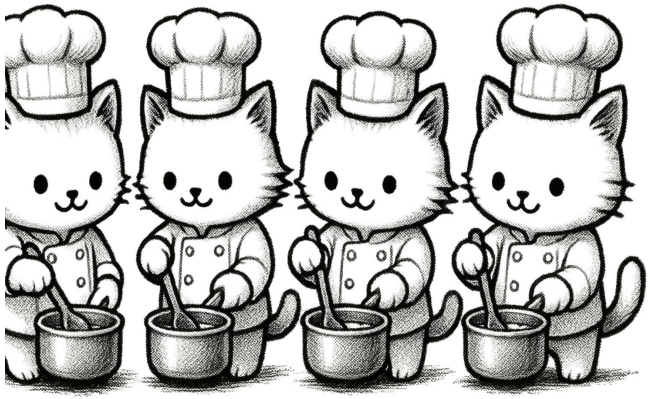
```
__device__ inline void storer(const globals &G, comp_sem &sem, comp_smem &smem, comp_regs &regs) {
    int signal_dev_idx = regs.task_id % NUM_DEVICES;
    device<NUM_DEVICES>::signal(G.barrier, {idx.x, idx.y}, signal_dev_idx, 1);
}
```

Signal after output tile store

```
__device__ inline void communicator(const globals &G, comm_sem &sem, comm_smem &smem, comm_regs &regs) {
    int2 idx = interpret_task(regs.task_id);
    if (threadIdx.x == 0)
        device<NUM_DEVICES>::wait(G.barrier, {idx.x, idx.y}, G.dev_idx, NUM_DEVICES);
    __syncthreads();
    group<NUM_WARPS>::all_reduce<ROW_BLOCK, COL_BLOCK, reduce_op::ADD>(G.C, {idx.x, idx.y});
}
```

Wait for signal, perform all-reduce

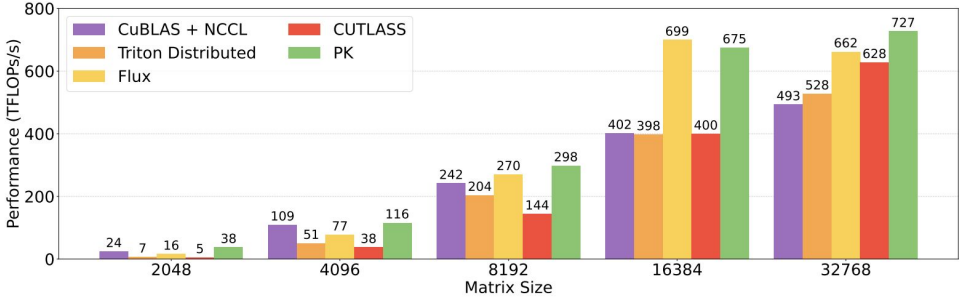
# Results



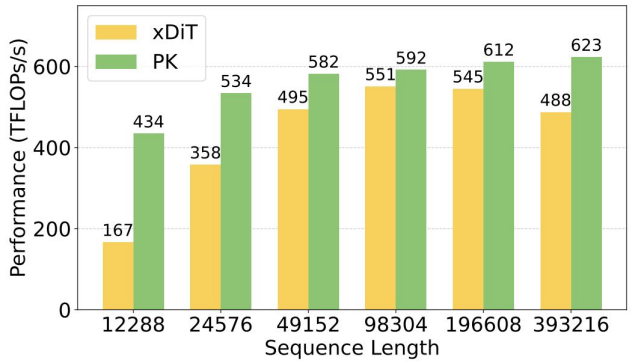
# PK Simplifies Multi-GPU Kernels

- Data and Tensor Parallelism
  - AllGather + GEMM, GEMM + ReduceScatter, GEMM + AllReduce
- Sequence Parallelism
  - Ring Attention, Ulysses Attention
- Expert Parallelism
  - MoE Token Dispatch + GEMM
- Collective Communication
  - AllReduce, AllGather, ReduceScatter, AllToAll

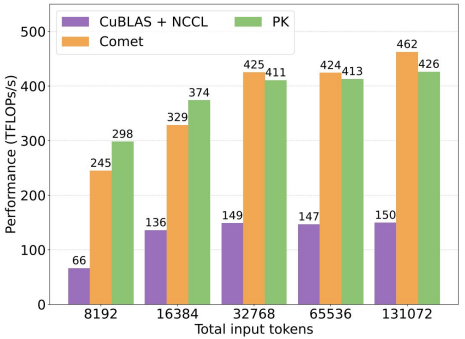
# PK Improves Performance, Reduces Complexity




## All-Gather GEMM (0.97x–7.39x)




## Ring Attention (1.07x–4.08x)




## MoE Dispatch + GEMM (0.92–1.22x)

# Adoption


# Conclusion

# Summary

Three key principles for multi-GPU kernel design:

- Transfer Mechanisms
- Scheduling
- Design Overheads

# Moving Forward

Infiniband / Ethernet



Multi-Silicon

# Thank You!

Paper

Code