Fast Multiplication in Binary Fields on GPUs via Register Cache

Mark Silberstein

Technion

Eli Ben-Sasson, **Matan Hamilis**, Eran Tromer

Brief

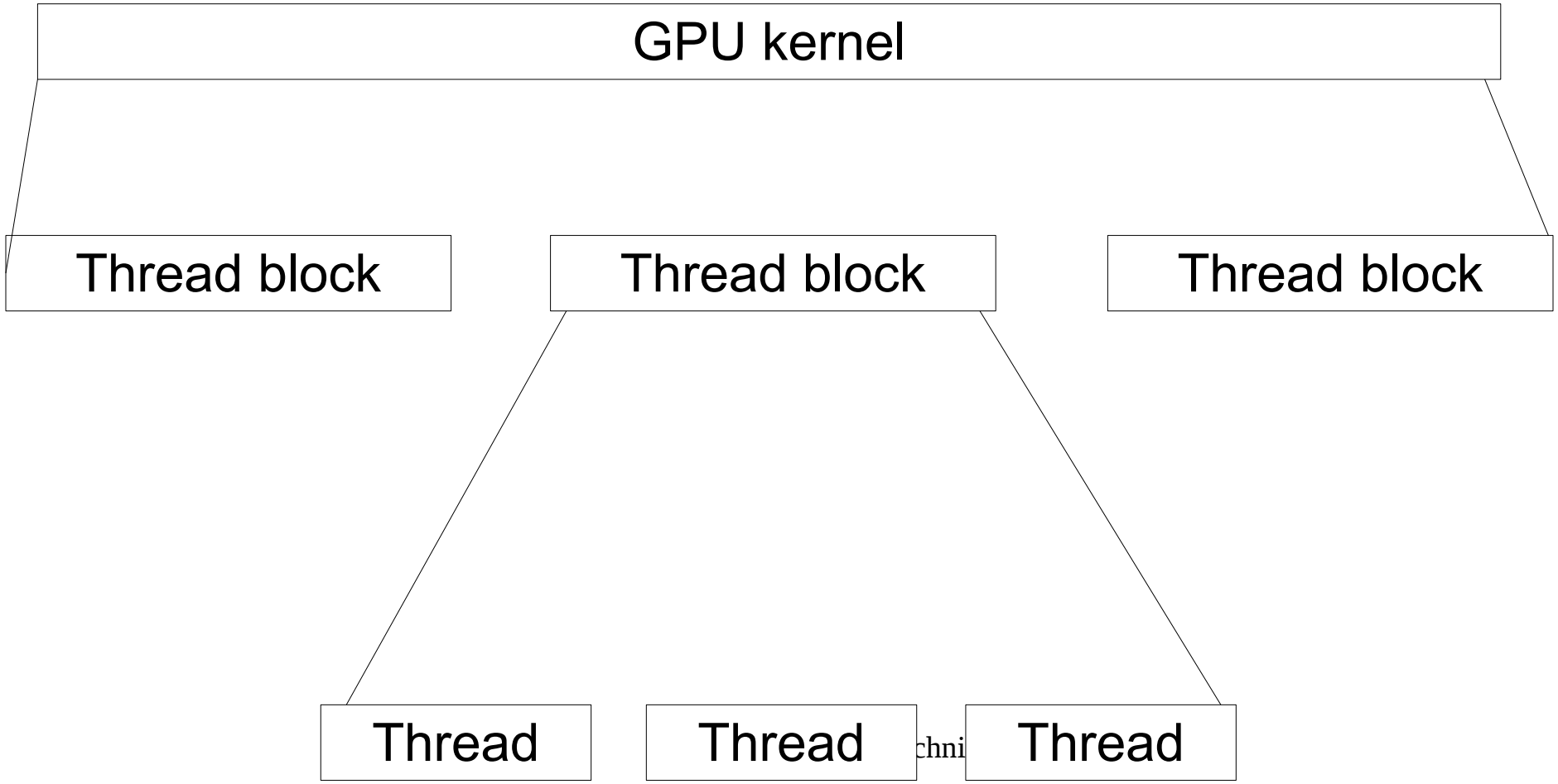
- Optimization methodology

Register cache: replace shared memory by **registers**

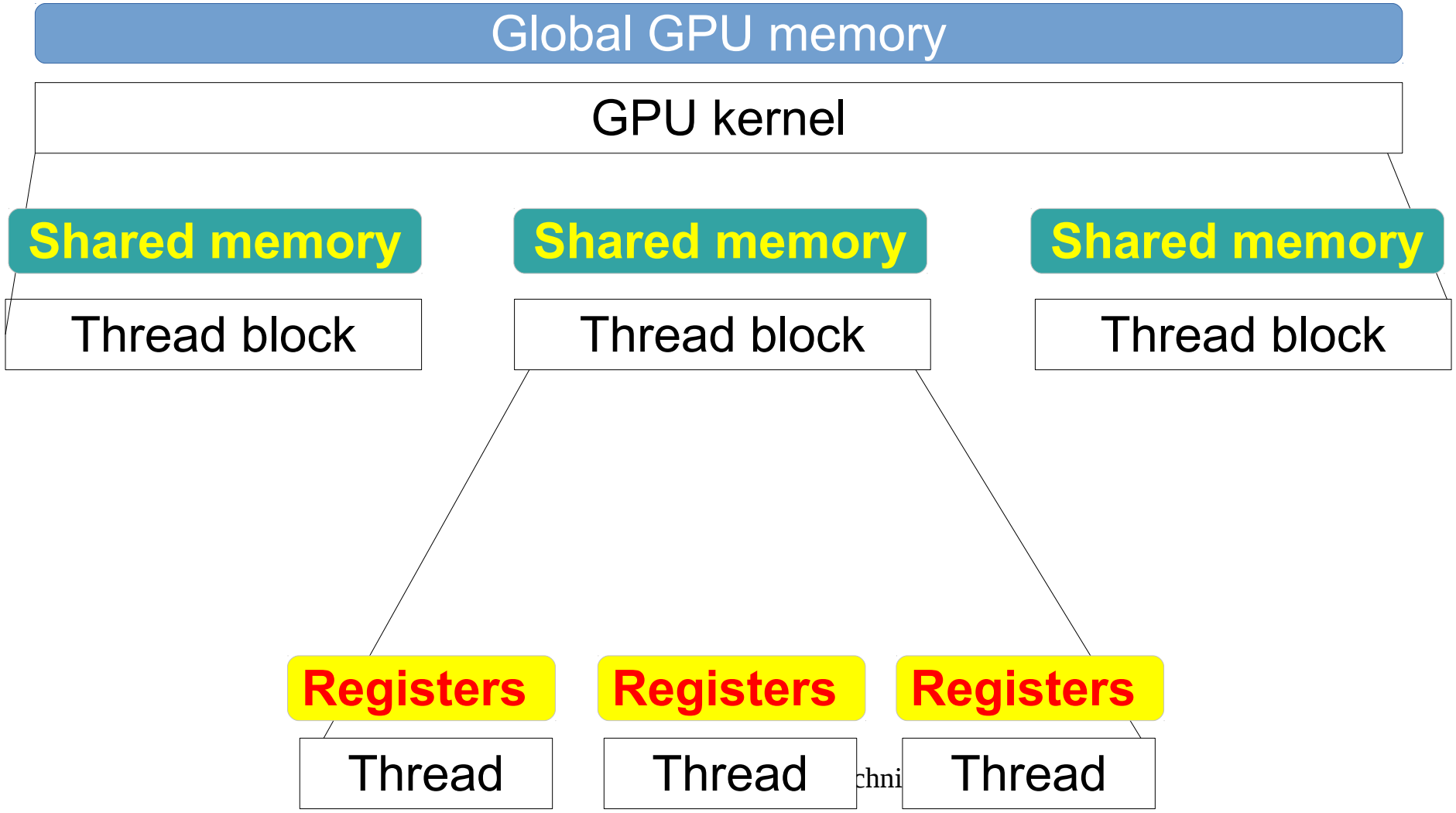
- Target applications: shared memory to cache input (e.g. stencil)
- **Our case: binary field multiplication**
- **Result: 50% speedup over baseline**

  x138 over a single core CPU with Intel's *CLMUL* instruction

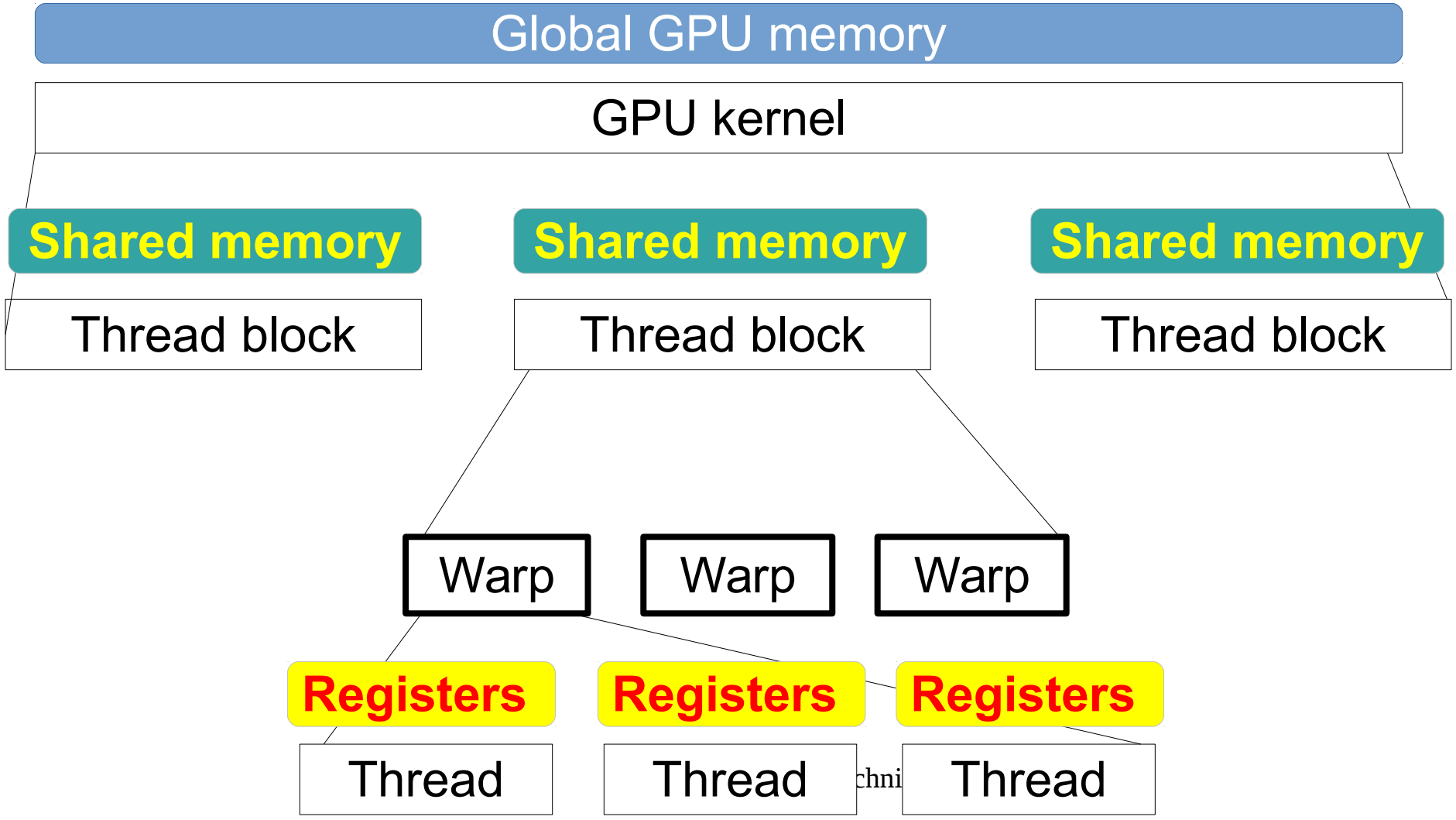
Background: execution hierarchy on NVIDIA GPUs



Background: memory and execution hierarchy on NVIDIA GPUs



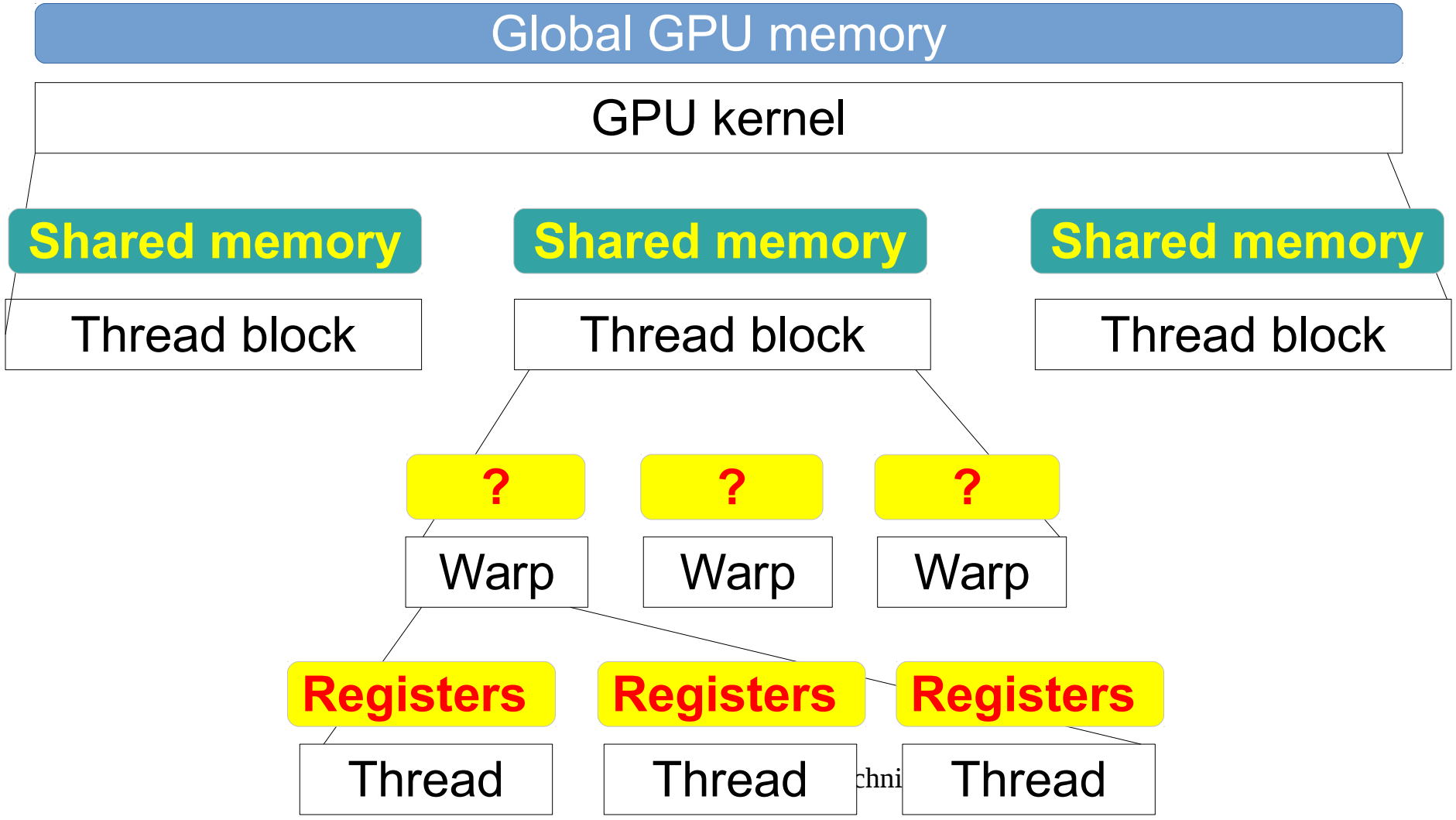
Warps: Not part of programming model



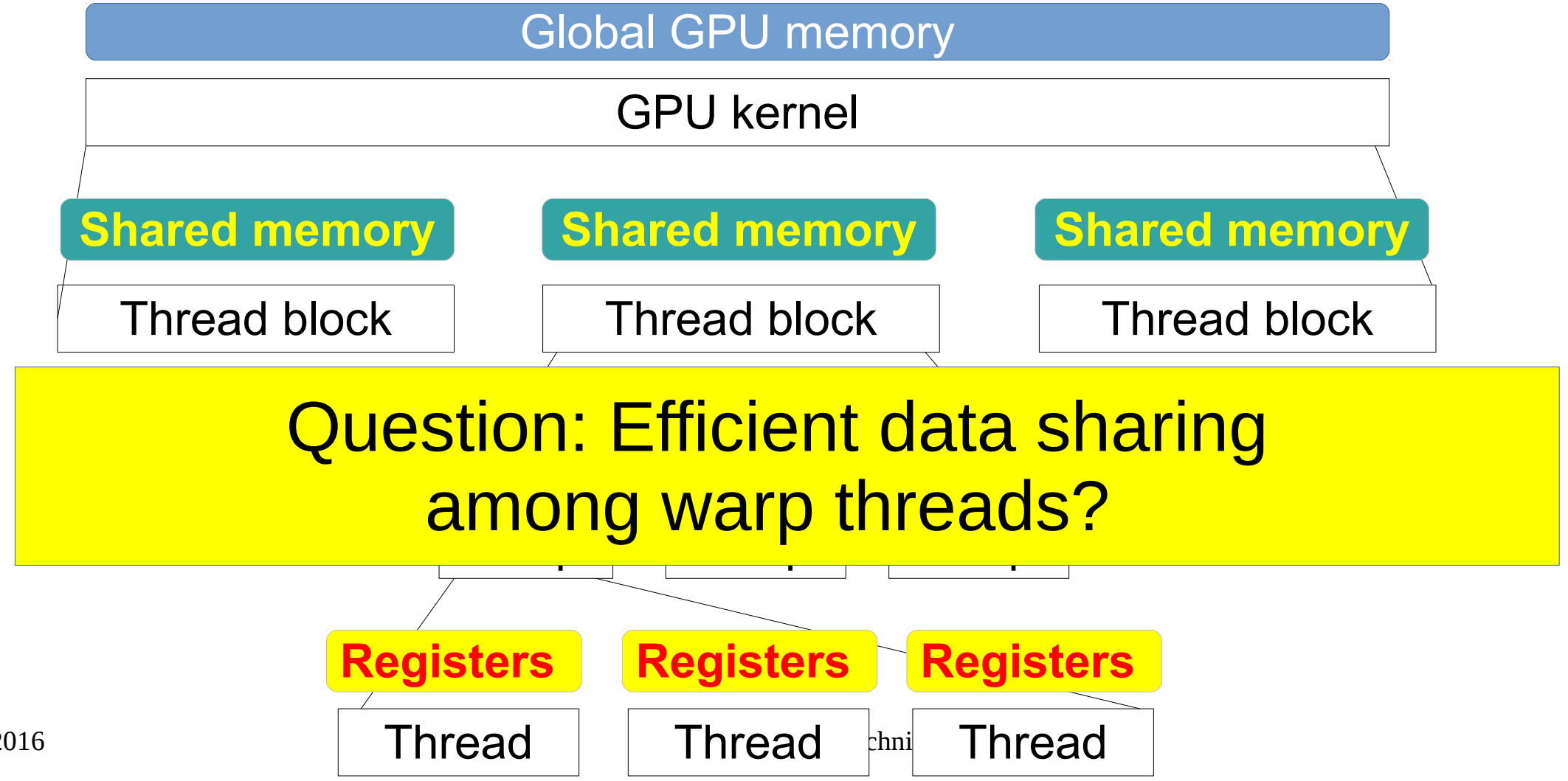
Why warp-centric programming

- MIMD divergence-free programming across warps
- SIMD-optimized lock-step execution
- “Free” synchronization among threads

Missing layer: warp cache?

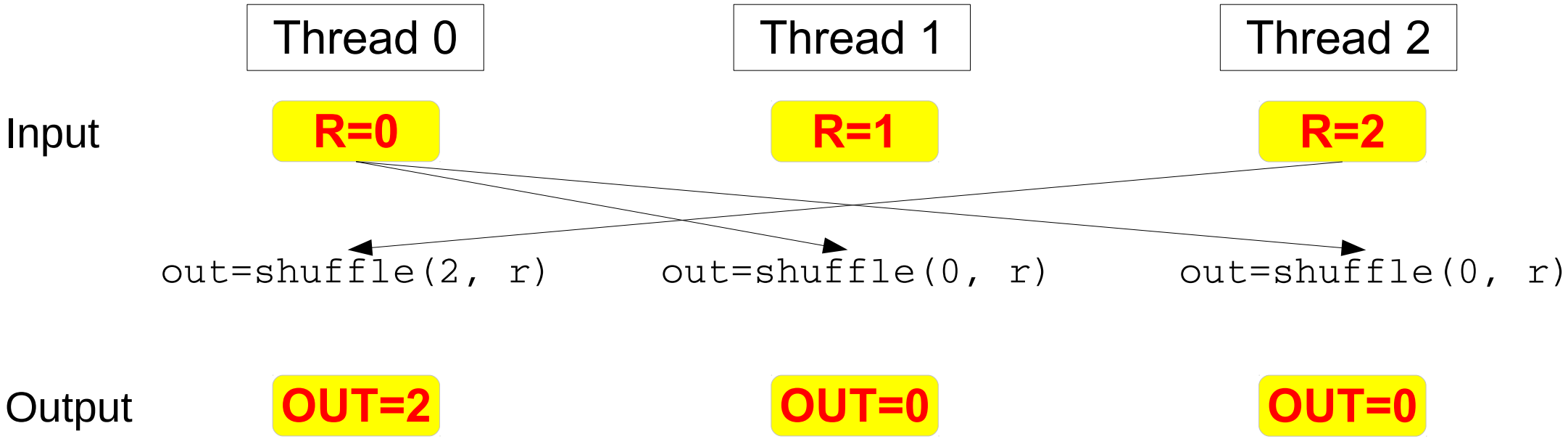


Missing layer: warp cache?



Shuffle: **warp-level** intrinsics
Reading other thread's registers

```
shuffle(SourceThreadID, OutputRegister)
```



Shuffle vs. shared memory

- No `__syncthreads` overhead
- Significantly higher bandwidth

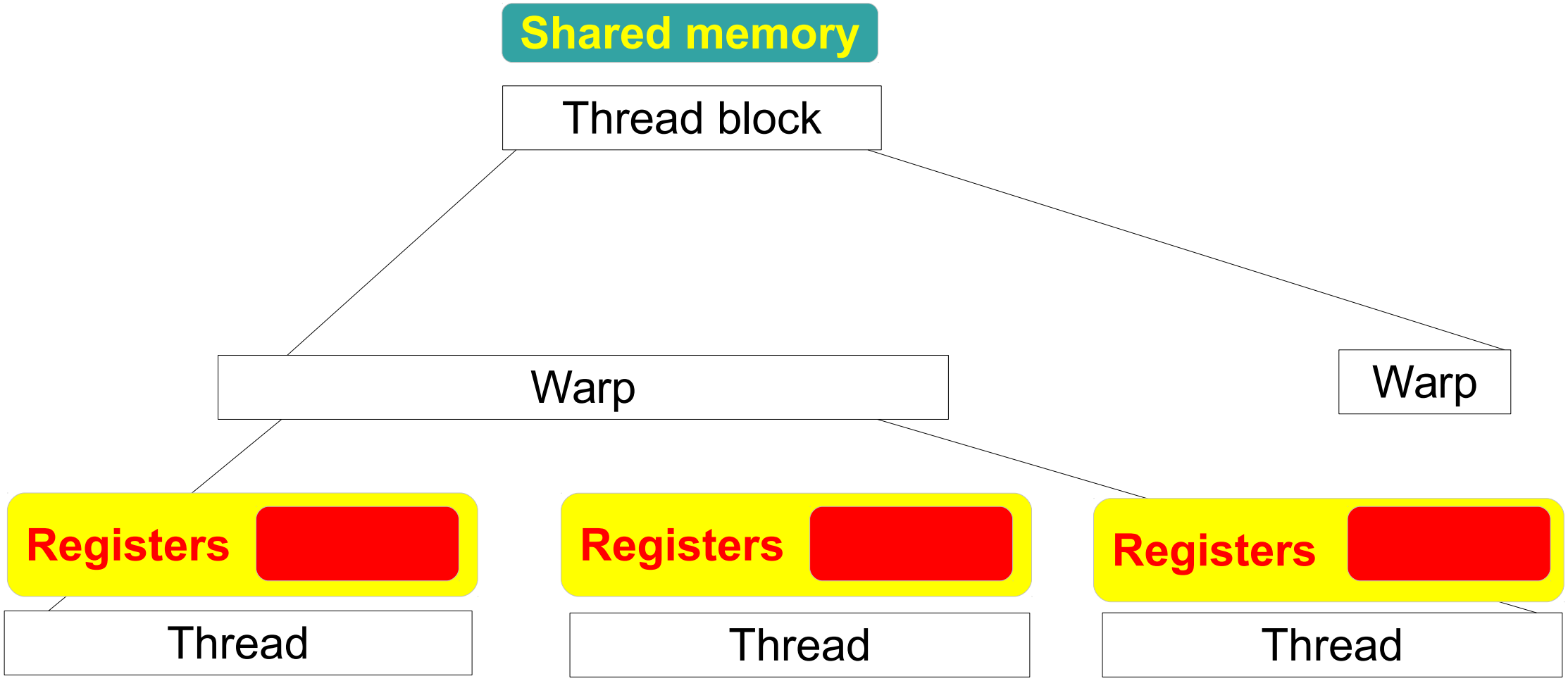
Shuffle vs. shared memory

- No `__syncthreads` overhead
- Significantly higher bandwidth

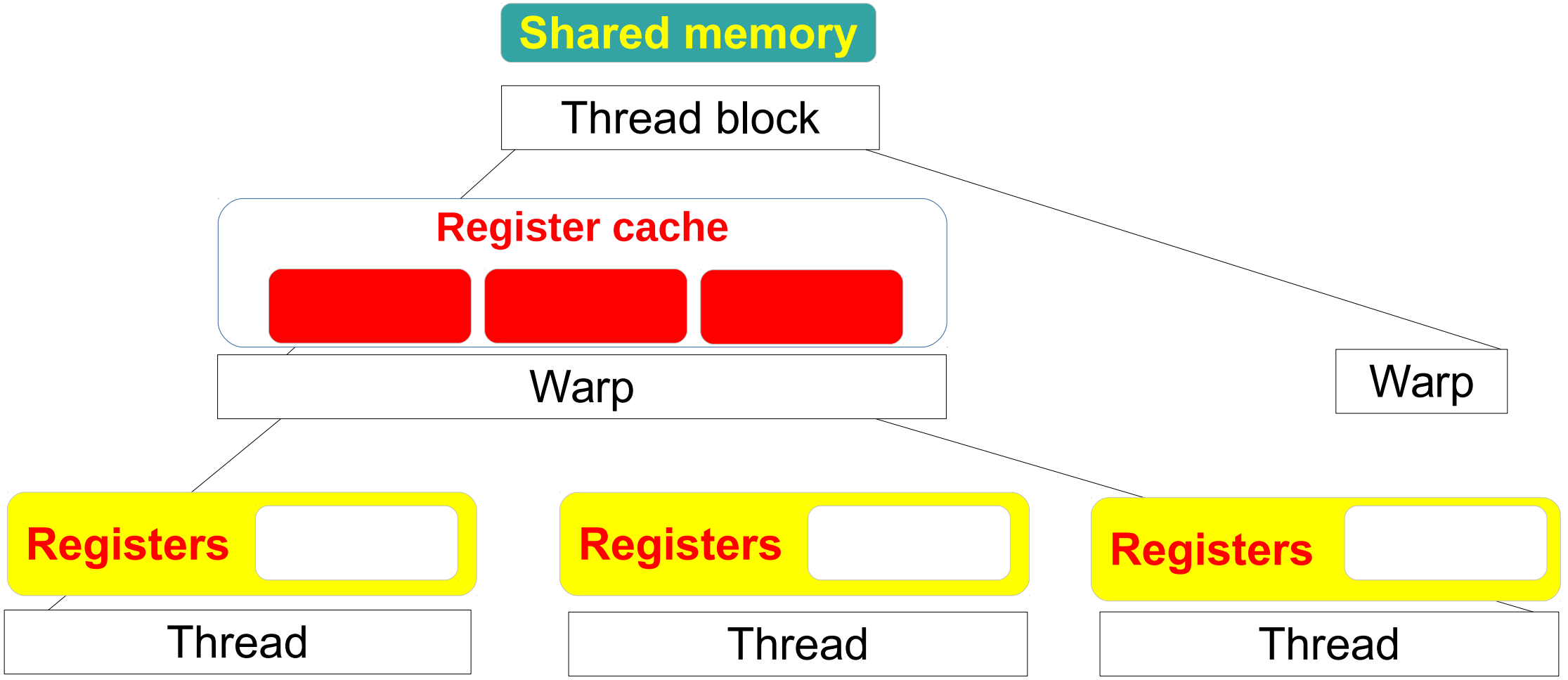
Challenge: programming complexity!

Application-specific algorithm modifications

This work: general *technique* to replace **input** shared memory with `shuffle`



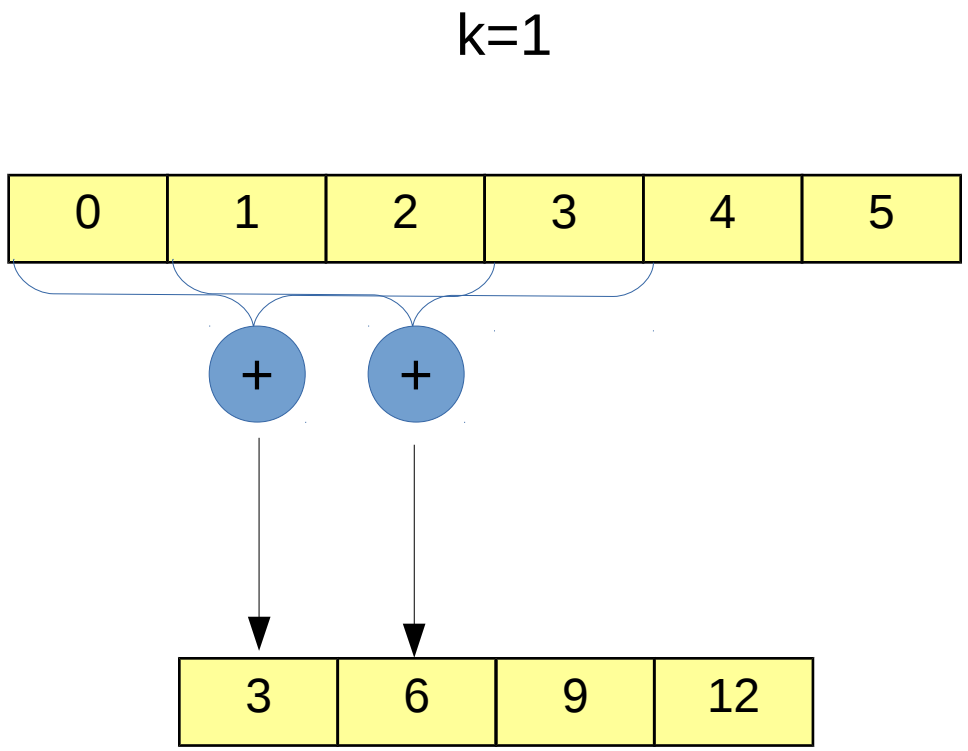
This work: general *technique* to replace **input** shared memory with `shuffle`



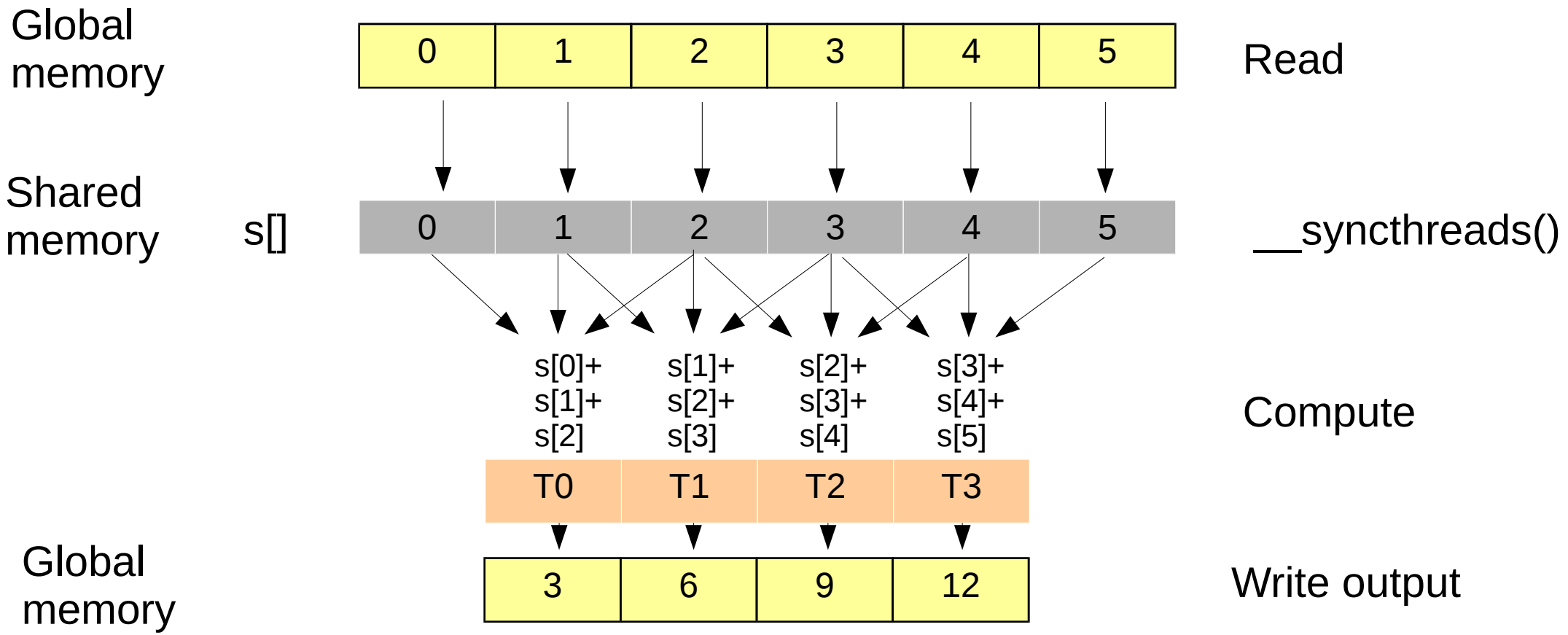
Outline

- Code transformation example: 1-d k-stencil
- General methodology
- Binary field multiplication
- Evaluation

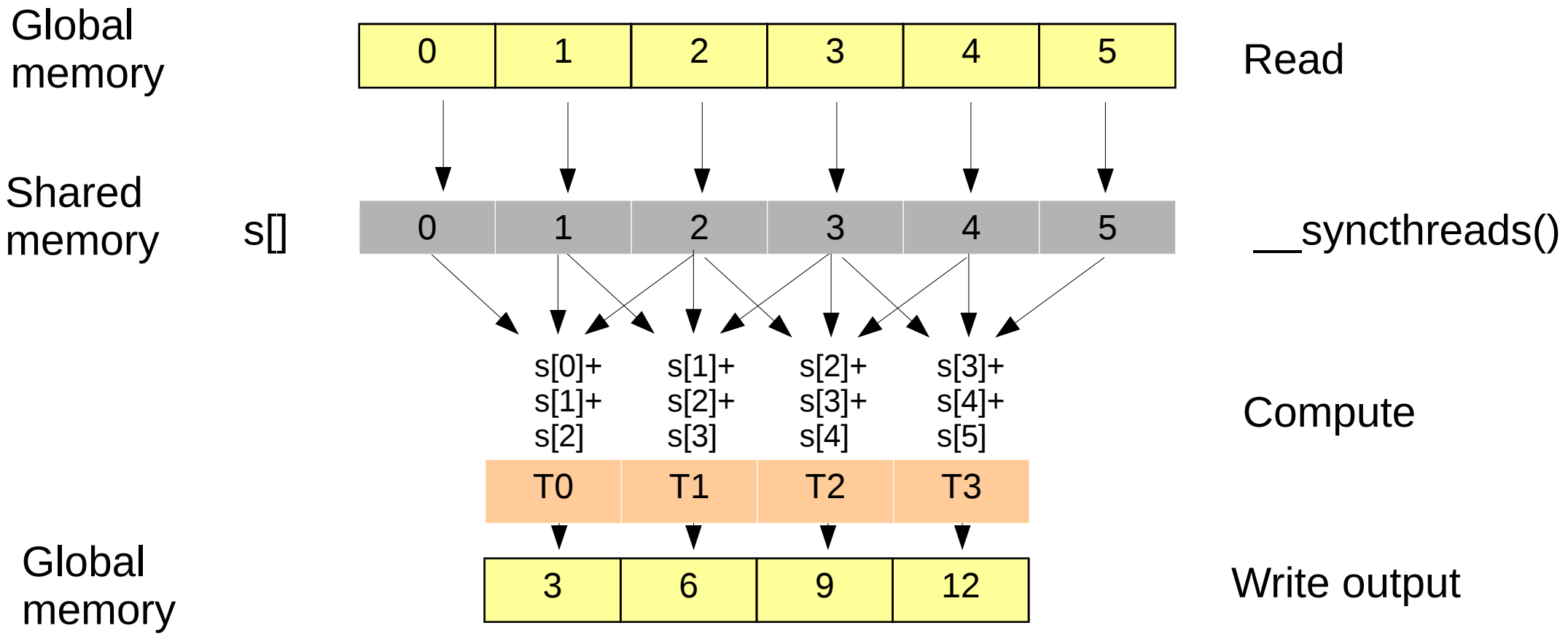
1-d k-stencil



1-d 1-stencil: shared memory



1-d 1-stencil: shared memory



Goal: eliminate shared memory access

1. Determine warp input

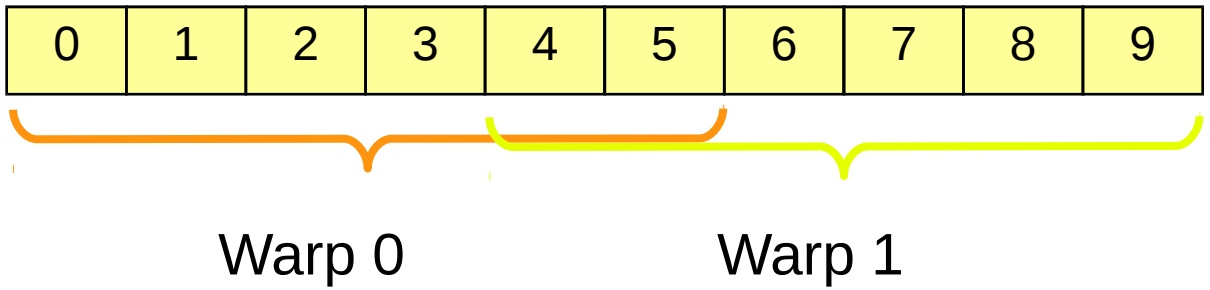
assume 4 threads/warp

Global memory input

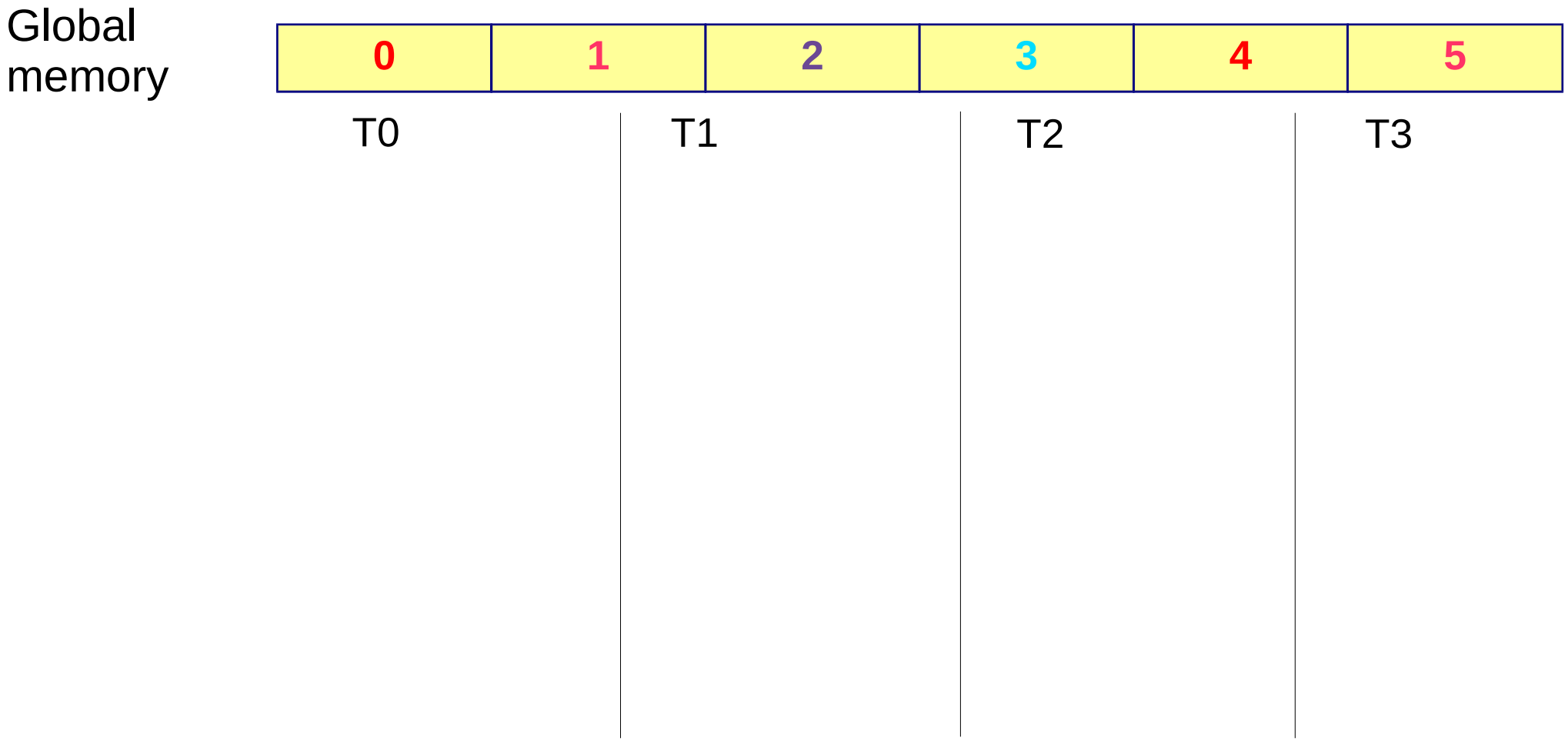
1. Determine warp input

assume 4 threads/warp

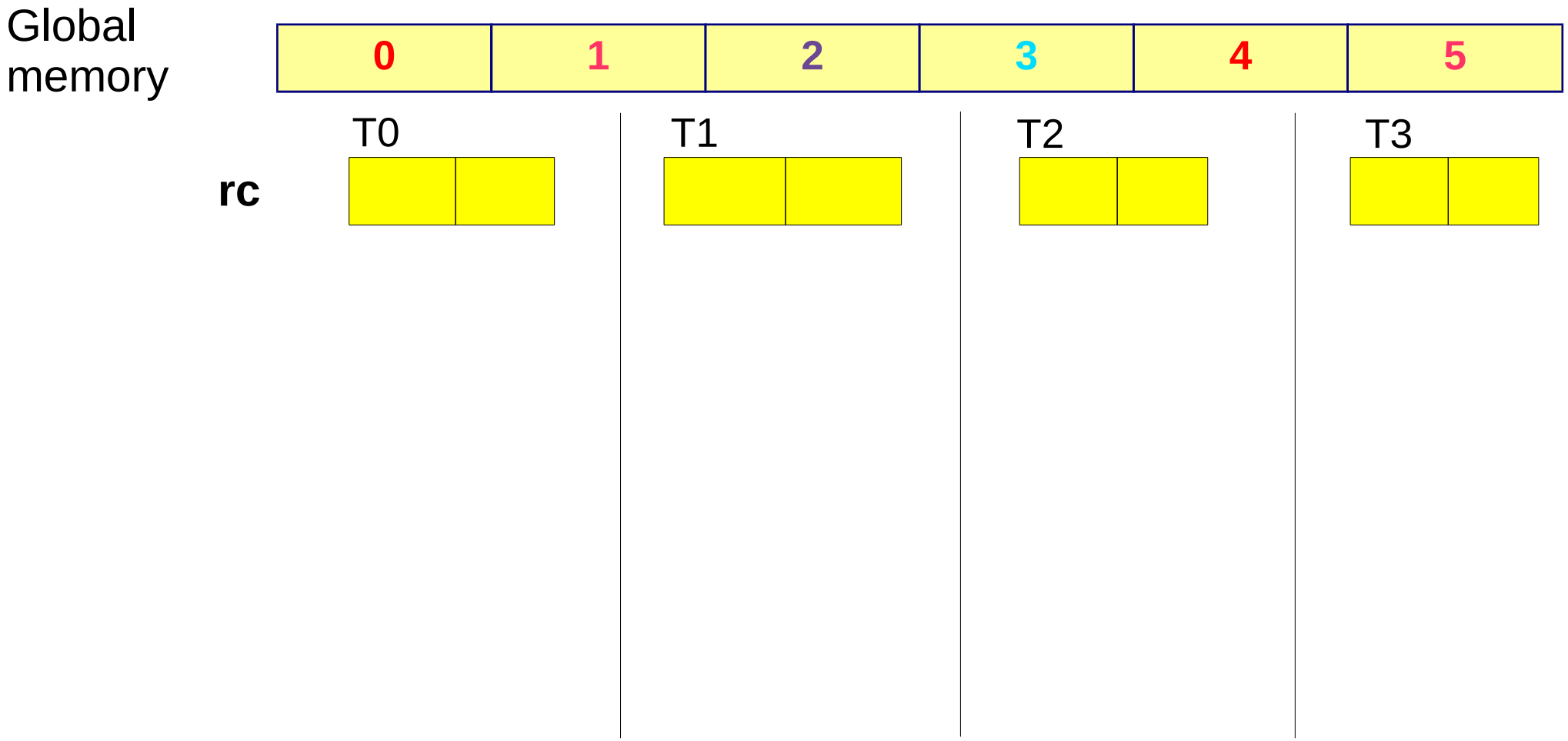
Global memory input



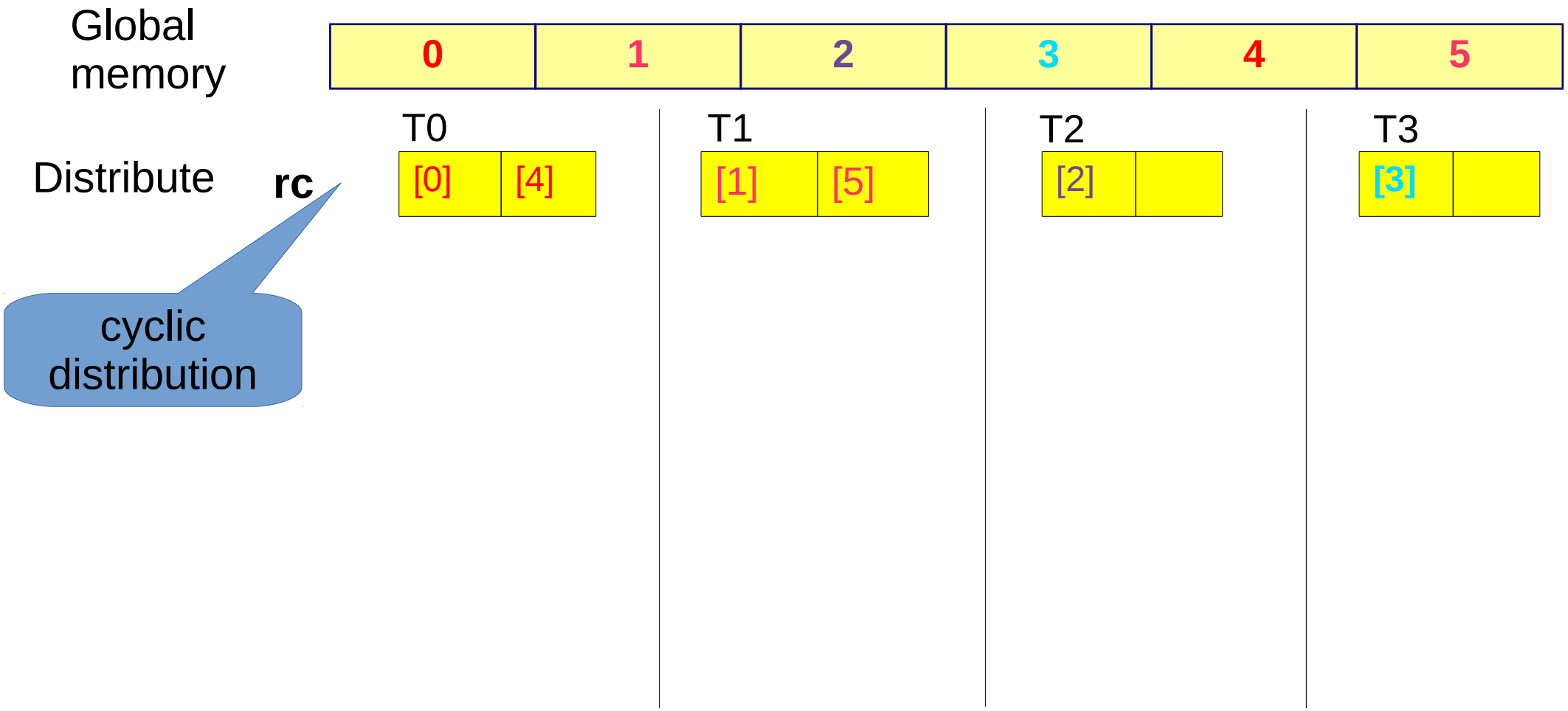
2. Assign input to *owner* thread



2. Assign input to *owner* thread



2. Assign input to *owner* thread



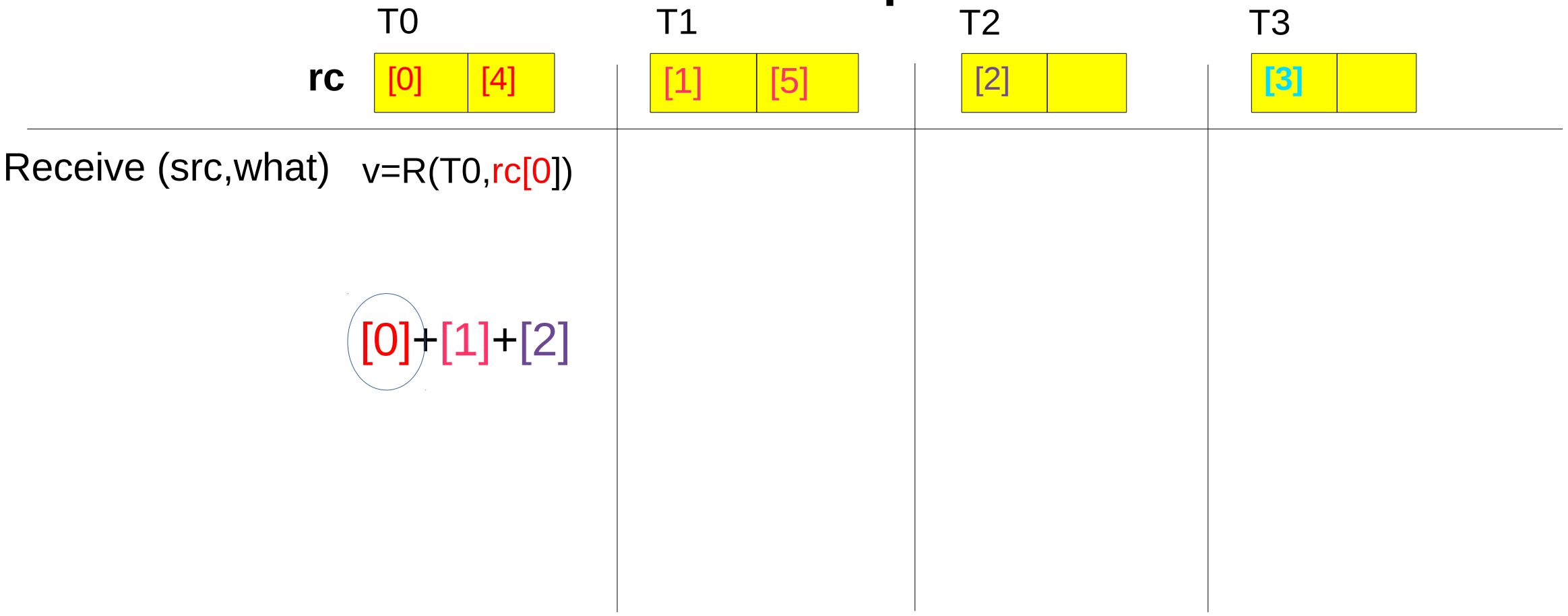
Some thread inputs are remote!

We define new communication primitives

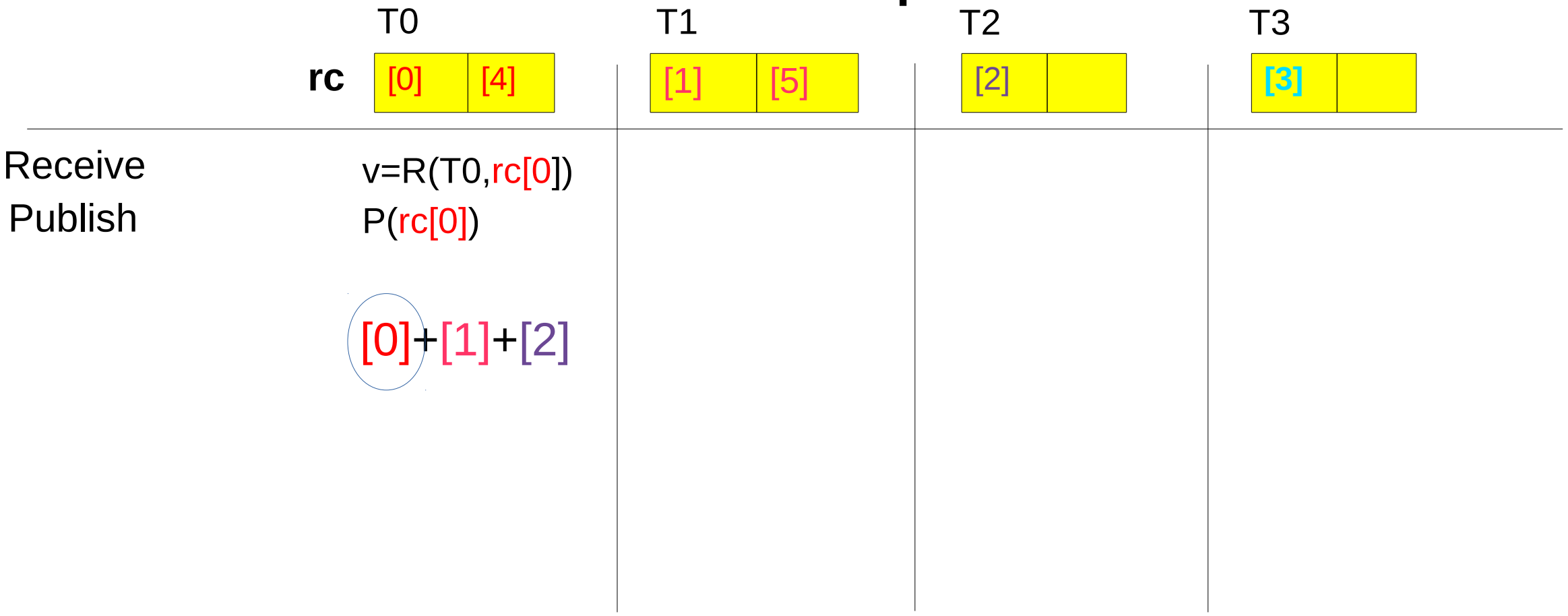
- **`Receive`**`(src_tid, remote_reg)` – receive data stored in thread `src_tid` in remote variable `remote_reg`
- **`Publish`**`(local_reg)` – publish local data stored in variable `local_reg`
- For one thread to `Receive`, another has to `Publish`!

2. Communication phase: Receive

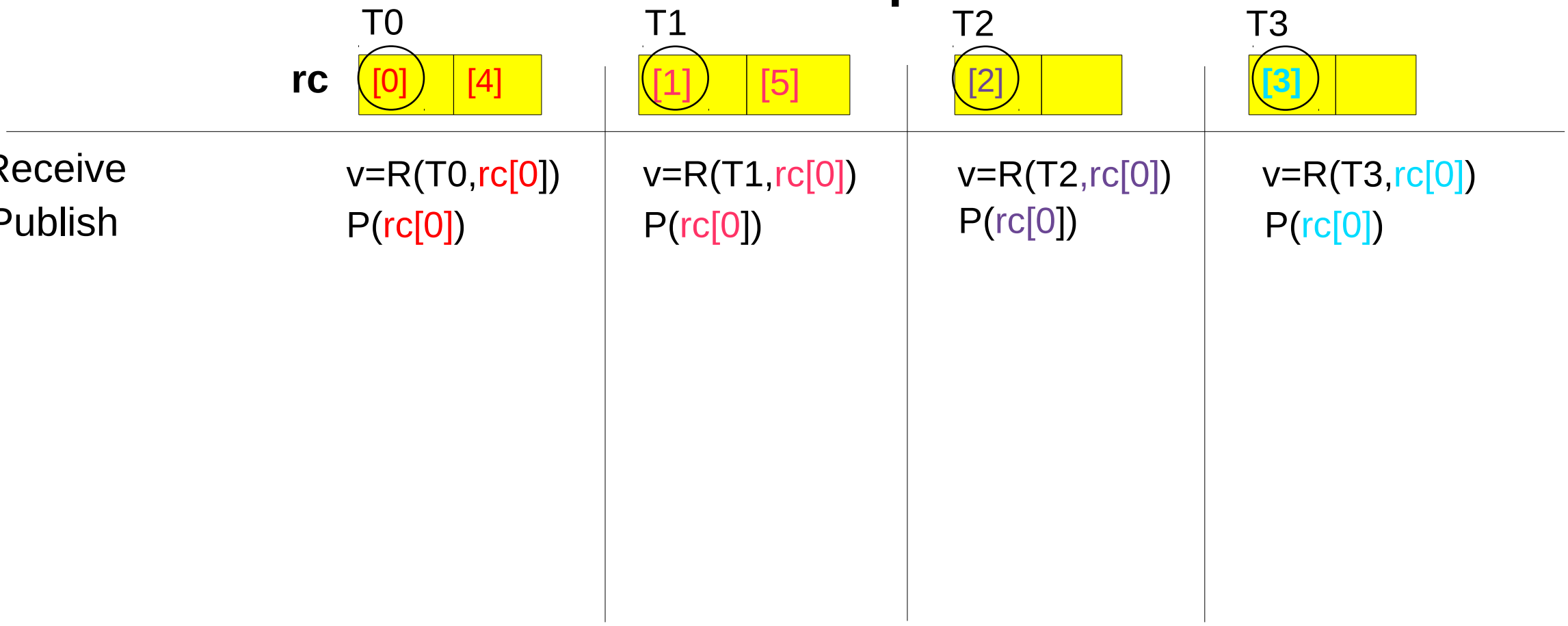
2. Communication phase: Receive



2. Communication phase: Publish



2. Communication phase: Publish



3. Computation phase

	T0	T1	T2	T3
rc	[0] [4]	[1] [5]	[2]	[3]
Receive (R) Publish (P)	v=R(T0,rc[0]) P(rc[0])	v=R(T1,rc[0]) P(rc[0])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v

_ac=[0], need [1]

2. Communication phase: Receive

	T0	T1	T2	T3
rc	[0] [4]	[1] [5]	[2]	[3]
Receive (R) Publish (P)	v=R(T0,rc[0]) P(rc[0])	v=R(T1,rc[0]) P(rc[0])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v
Receive	v=R(T1,rc[0])	v=R(T2,rc[0])	v=R(T3,rc[0])	**v=R(T0,rc[1])**

2. Communication phase: Publish

	T0	T1	T2	T3
rc	[0] [4]	[1] [5]	[2]	[3]
Receive (R) Publish (P)	v=R(T0,rc[0]) P(rc[0])	v=R(T1,rc[0]) P(rc[0])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v
Receive Publish	v=R(T1,rc[0]) P(rc[1])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])	**v=R(T0,rc[1])** P(rc[0])

3. Computation phase

	T0	T1	T2	T3
rc	[0] [4]	[1] [5]	[2]	[3]
Receive (R) Publish (P)	v=R(T0,rc[0]) P(rc[0])	v=R(T1,rc[0]) P(rc[0])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v
Receive Publish	v=R(T1,rc[0]) P(rc[1])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])	**v=R(T0,rc[1])** P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v

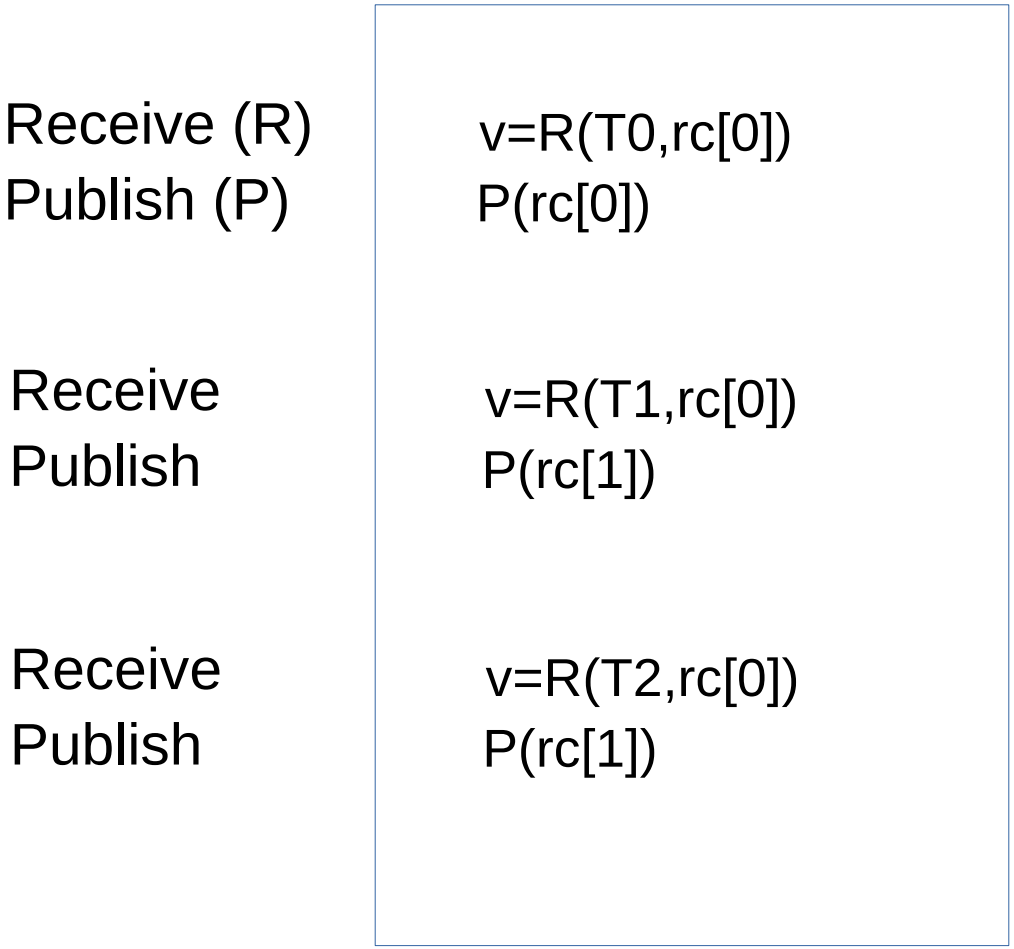
_ac=[0]+[1], need [2]

4. write result to global memory

	T0	T1	T2	T3
rc	[0] [4]	[1] [5]	[2]	[3]
Receive (R) Publish (P)	v=R(T0,rc[0]) P(rc[0])	v=R(T1,rc[0]) P(rc[0])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v
Receive Publish	v=R(T1,rc[0]) P(rc[1])	v=R(T2,rc[0]) P(rc[0])	v=R(T3,rc[0]) P(rc[0])	v=R(T0,rc[1]) P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v
Receive Publish	v=R(T2,rc[0]) P(rc[1])	v=R(T3,rc[0]) P(rc[1])	**v=R(T0,rc[1])** P(rc[0])	**v=R(T1,rc[1])** P(rc[0])
Compute	_ac+=v	_ac+=v	_ac+=v	_ac+=v

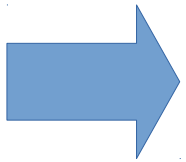
_ac=[0]+[1]+[2]

Receive + Publish = `shuffle`

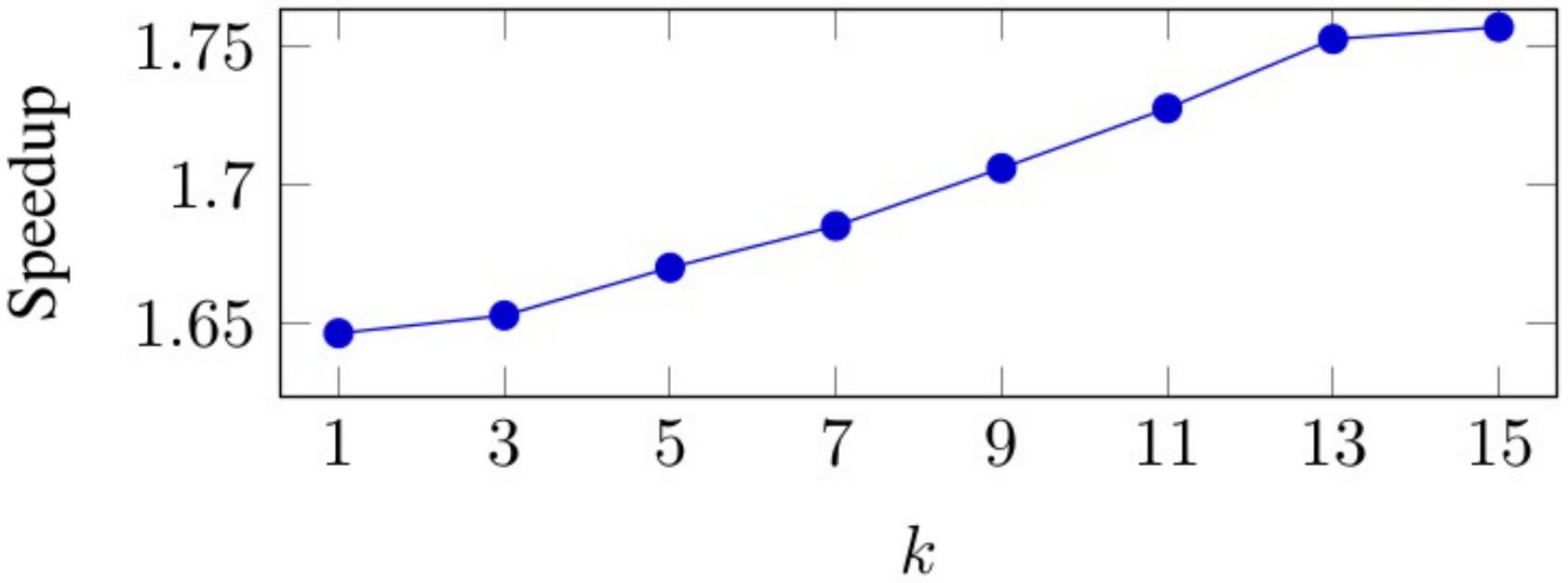


Receive + Publish = `shuffle`

Receive (R) Publish (P)	v=R(T0,rc[0]) P(rc[0])	pub_idx=0;src=0; v=**shuffle**(src,rc[pub_idx])
Receive Publish	v=R(T1,rc[0]) P(rc[1])	pub_idx=1;src=1; v=**shuffle**(src,rc[pub_idx])
Receive Publish	v=R(T2,rc[0]) P(rc[1])	pub_idx=1;src=0; v=**shuffle**(src,rc[pub_idx])



Performance benefits for k-stencil



Up to 76%!

See the paper for further analysis for benefits and limitations

Summary: Register Cache

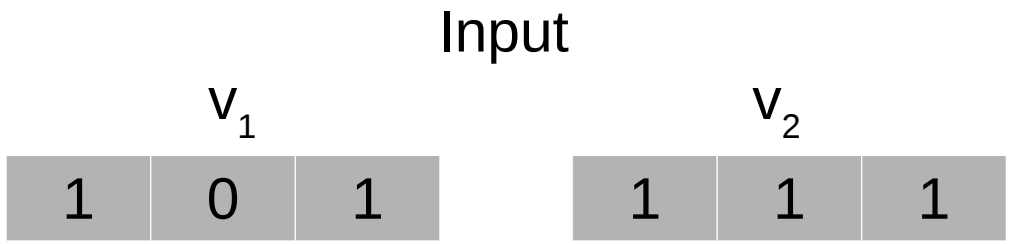
- Start from shared memory-based implementation
- Identify input for each warp
- Distribute data among threads
- Split in multiple phases
  - Communication phase: Publish – Receive
  - Computation phase
- Transform Publish-Receive into *shuffle*

Part 2: multiplication in large binary fields 2^n
$32<n<256$

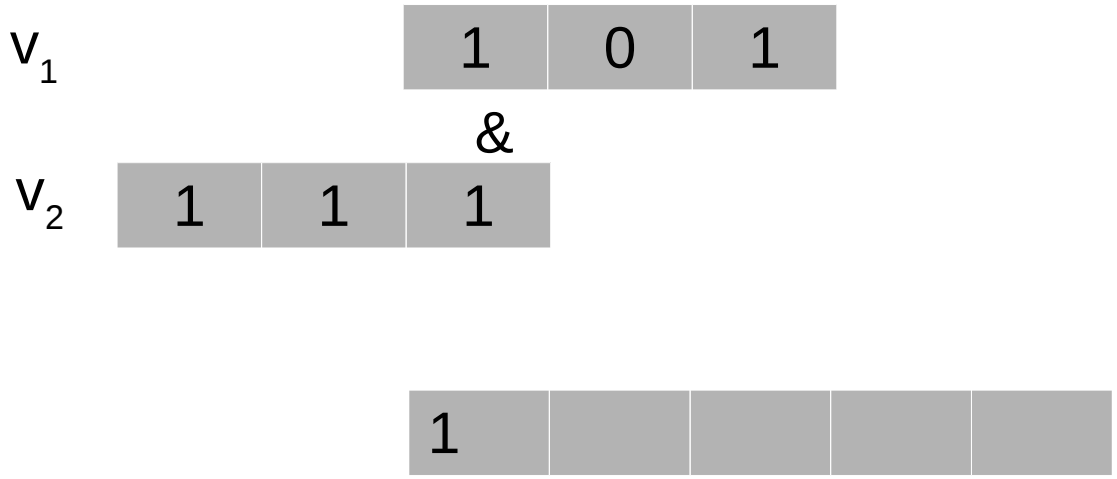
- Binary field multiplication – computational bottleneck in many applications
  - Security, Storage
- Typical scenario: multiply many pairs

- Main kernel: convolution of **binary** vectors of size *n*
- x86 CPUs: special CLMUL instruction
  - IvyBridge: 14 cycles, 2 convolutions

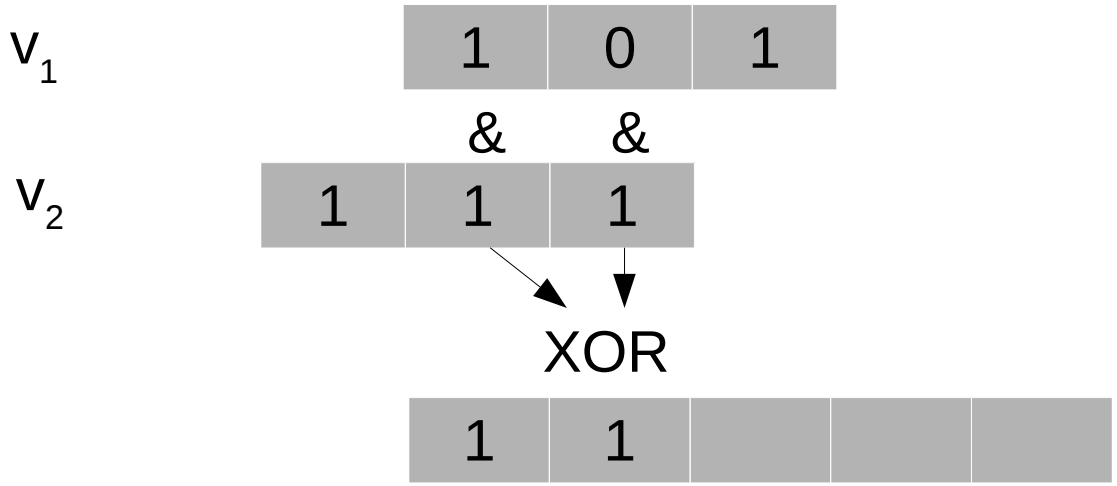
Binary convolution



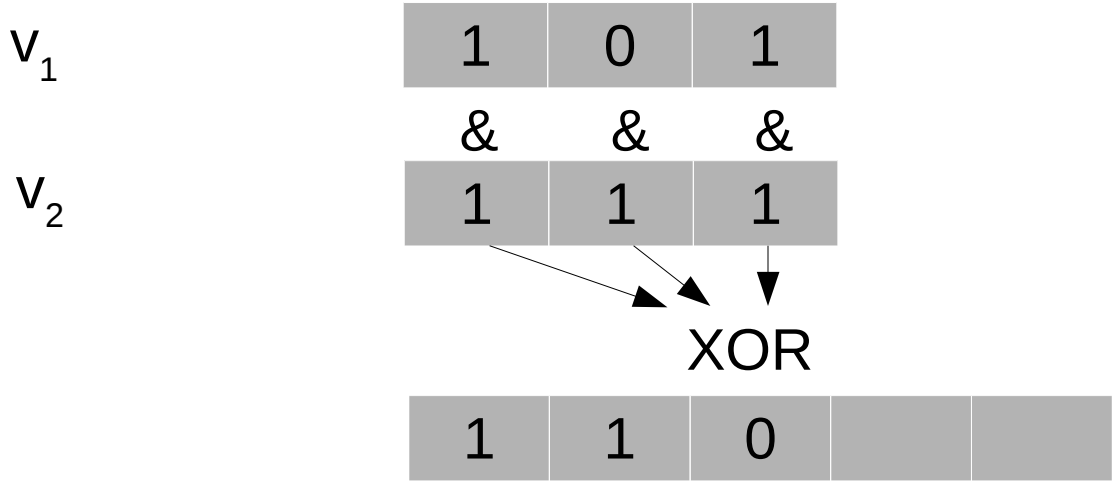
Binary convolution



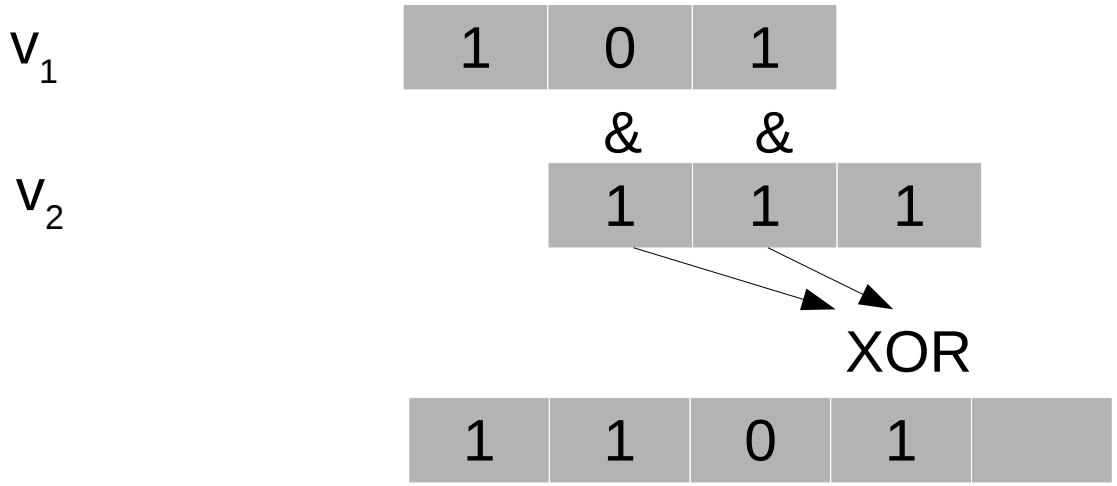
Binary convolution



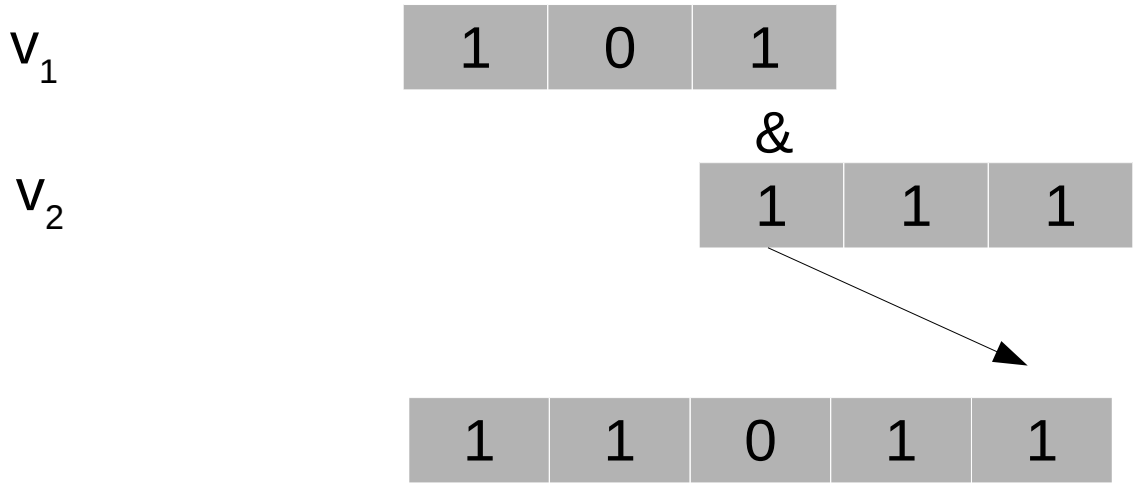
Binary convolution



Binary convolution



Binary convolution



Challenges - Solutions

- Bit-level operations

- Load balancing
  between warp threads
- Scaling to large fields

Challenges - Solutions

- Bit-level operations
- Load balancing between warp threads
- Scaling to large fields

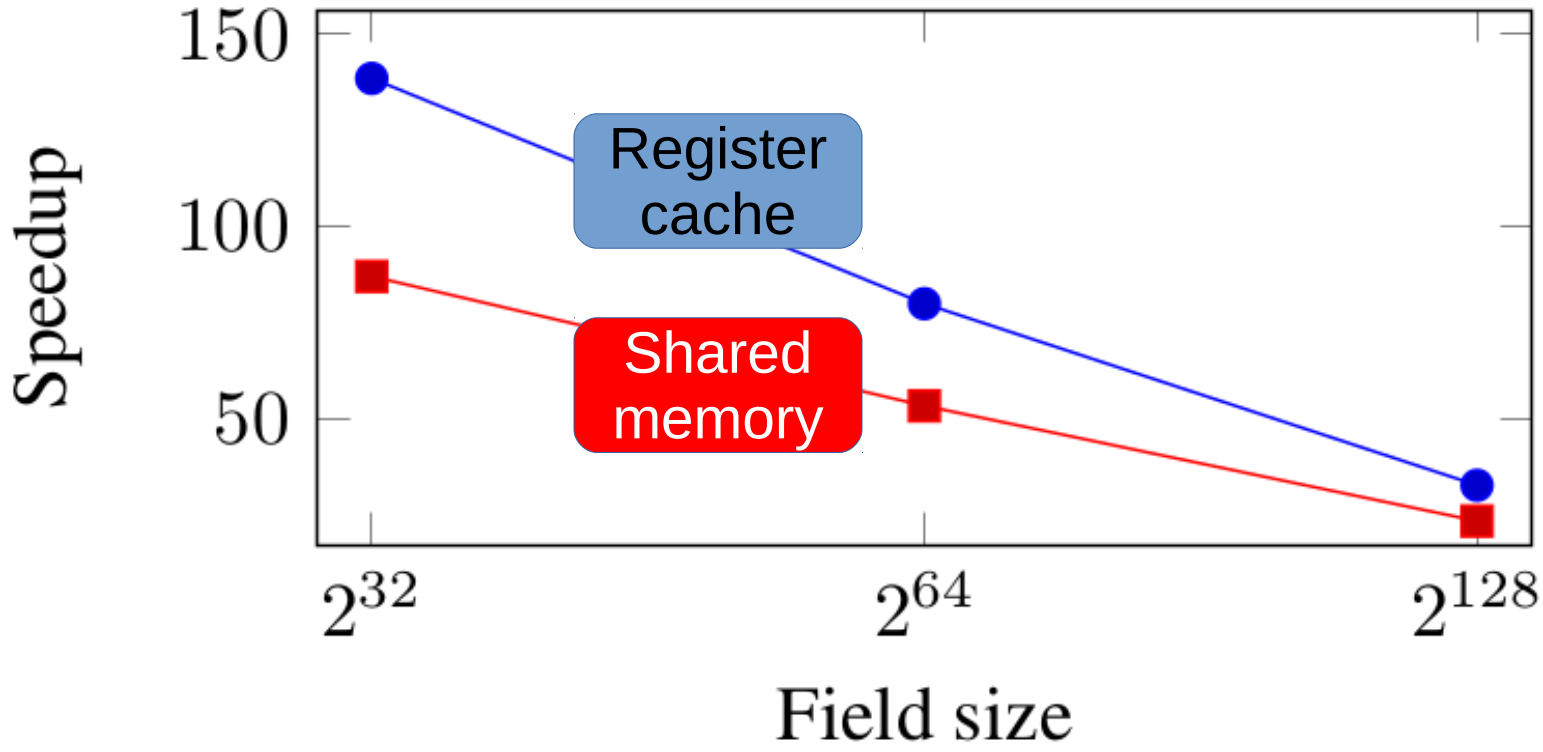
- **Bit slicing**

  Compute 32 convolutions in a single thread
- **Algorithmic trick** to achieve divergent free execution
- Use **register cache to free shared memory and scale better**

See the paper for details

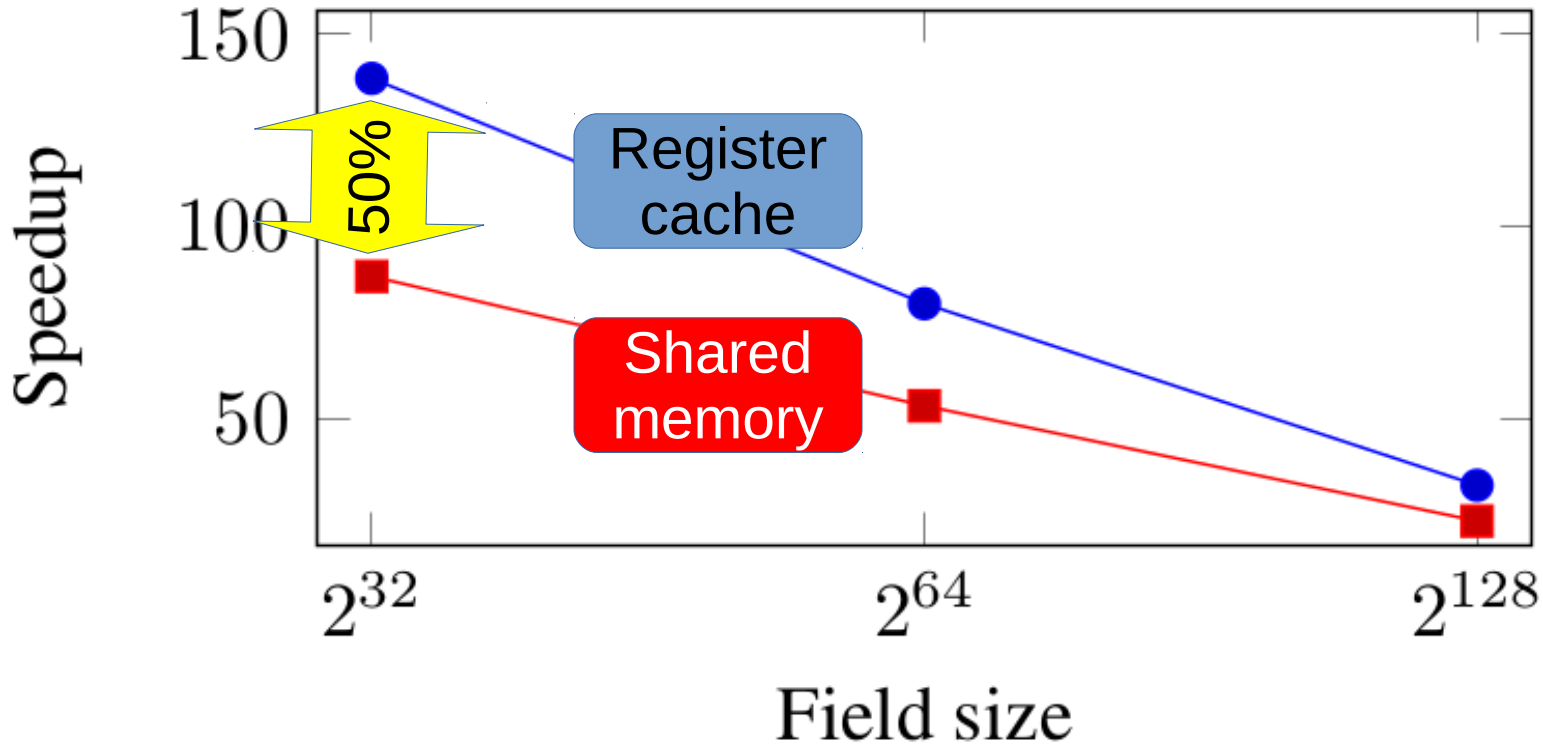
Performance

- CPU baseline: **CLMUL intrinsic** (via popular NTL library)
- NVIDIA K80: 138x faster than CPU



Performance

- CPU baseline: **CLMUL intrinsic** (via popular NTL library)
- NVIDIA K80: 138x faster than CPU



Conclusions

- Register cache: general technique for replacing shared memory with shuffle
- Apply to fast binary field multiplication
- Register cache improved application performance by 50%
- Total: x138 over CPU CLMUL for fields of size 32

Source code: https://github.com/HamilM/GpuBinFieldMult

Further questions: mark@ee.technion.ac.il