Instruction Set Extensions for Photonic Synchronous Coalesced Access

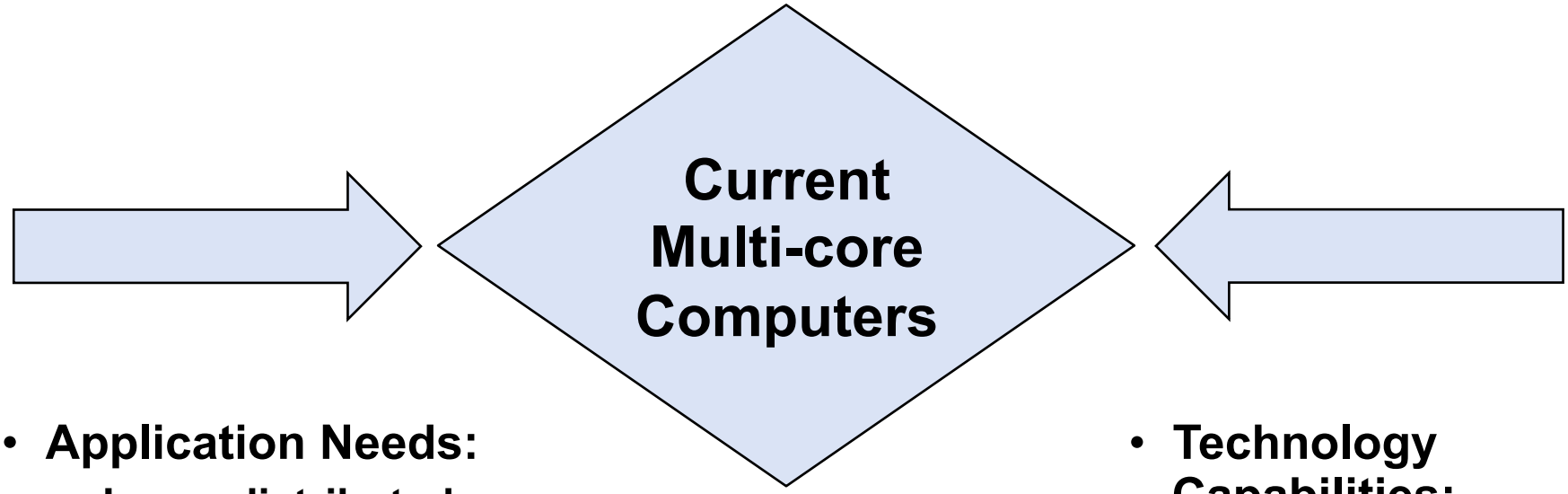
Paul Keltcher, David Whelihan, Jeffrey Hughes

September 12, 2013

LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

This work is sponsored by Defense Advanced Research Projects Agency (DARPA) under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government. Distribution Statement A. Approved for public release; distribution is unlimited.

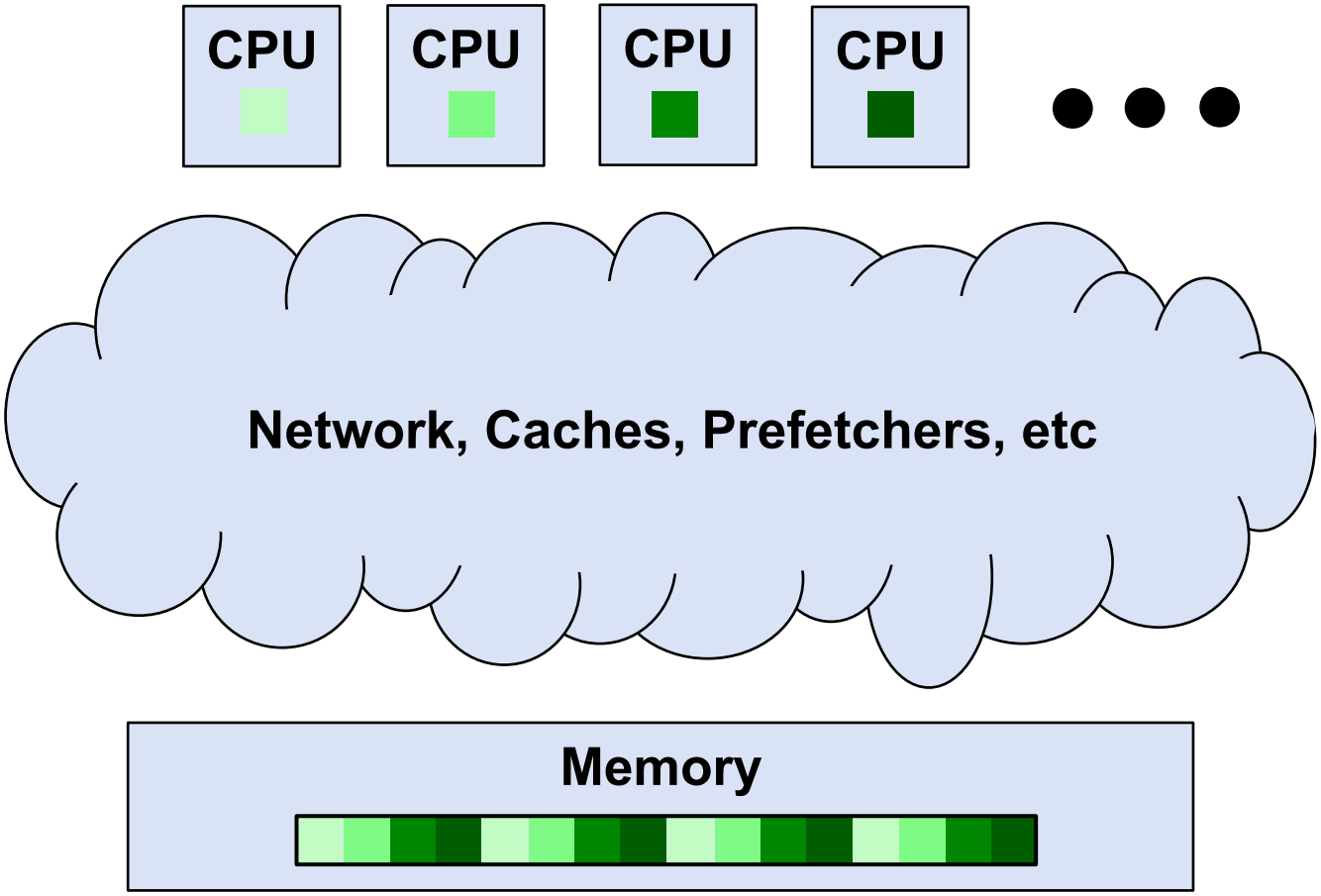
Trends



- **Application Needs:**
  - Large distributed data sets
  - Algorithm complexity
  - Parallel processing

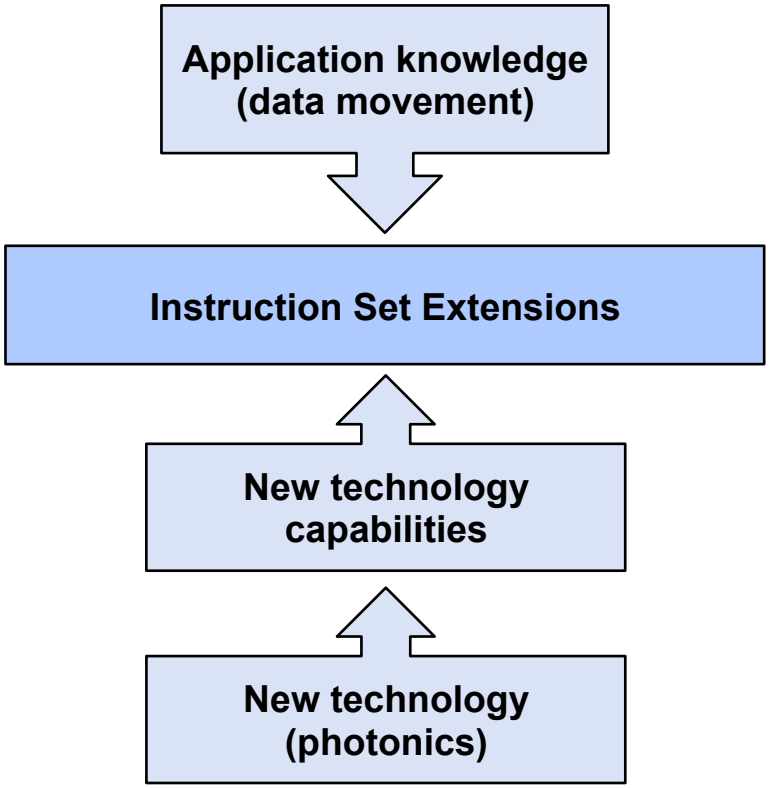
- **Technology Capabilities:**
  - No more frequency scaling
  - More cores
  - Distributed caches

Current Architectures



Lack of global coordination hurts both performance and power

Our Approach



- Need better synchronization mechanisms.
- Two components
  - Exploit application knowledge
  - Technology enabler
- Our Approach
  - Extended ISA
  - Utilize Photonics

This talk is about employing new technologies that enable us to program parallel processors easily and efficiently.

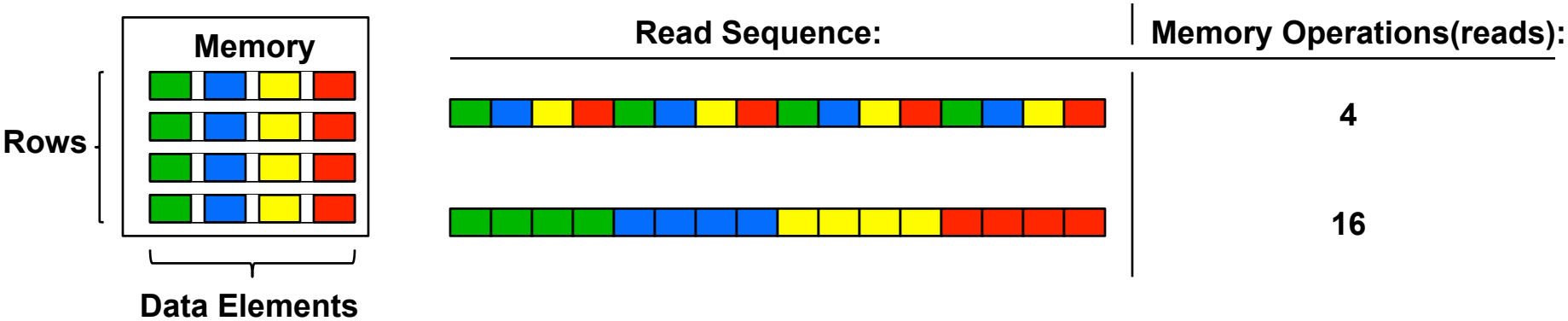
Outline

- Background
- The Multi-processor ISA
- Matrix transpose example
- Conclusions

Memory Access

Memory is a 1-dimensional resource, best accessed in contiguous chunks

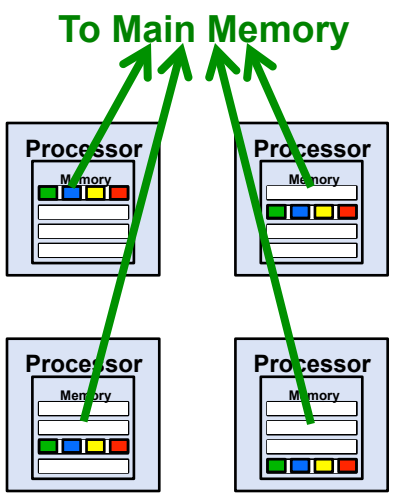
- **Physical memories are 2-d arrays that are accessed from only one dimension**
  - **To read a memory element, an entire row in the physical array must be read**
  - **Memory is most efficiently accessed sequentially**



Distributed Memory Access

Compute parallelism can increase performance, but it can also greatly exacerbate non-local data access problems

- **Restructuring of data can be extremely performance limiting**
- **Micro-architecture innovations can hinder efficient distributed data movement**



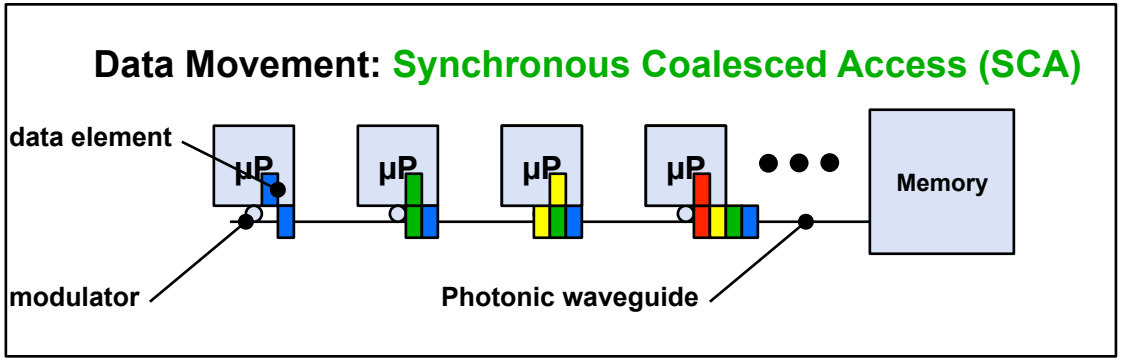
Read Sequence:	Memory Operations (reads):	Network Transfers:
	4	4
	16	16

Chip-scale Photonic Technology

- In our work we use chip-scale photonics to tightly integrate processors and memory
- Chip-scale photonic links are distance independent
- Photonics enables synchronization at long distance

Photonics enables scalable, global, synchronous communication

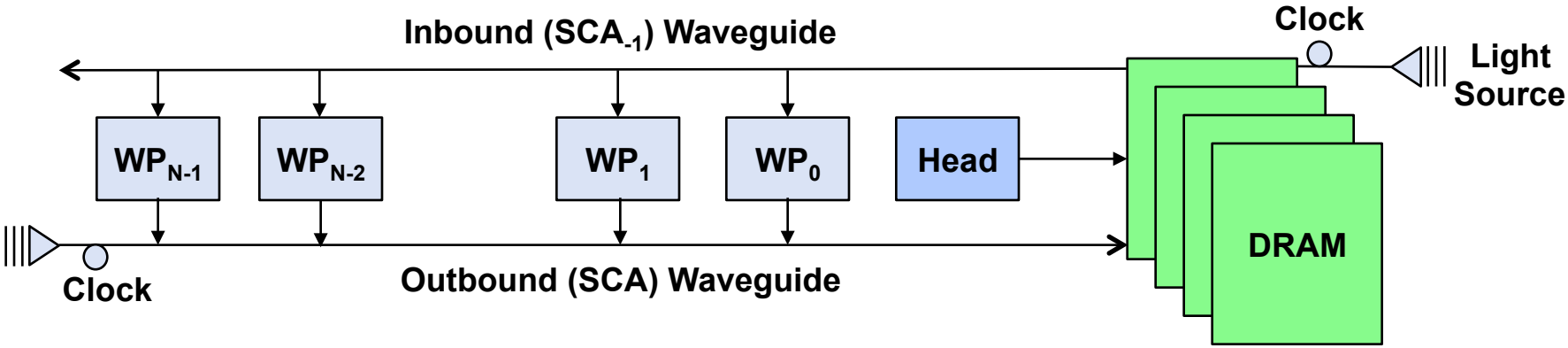
The Synchronous Coalesced Access (SCA)*



- The distance independent nature of photonics can be used to synchronize transfers from spatially separate chips or regions on chips
- Multiple independent data transfers synthesized on-the-fly
- This coordination can result in long ordered streams of data sourced from multiple locations
  - Globally synthesized accesses

* IPDPS 2013 "P-sync: A Photonically Enabled Architecture for Efficient Non-local Data Access"

P-sync Architecture*



- **Worker Processors share two TDM silicon photonic waveguides.**
- **The Head Node coordinates memory traffic for Worker Processors by issuing Memory Read/Write requests.**

*** IPDPS 2013 "P-sync: A Photonically Enabled Architecture for Efficient Non-local Data Access"**

Outline

- Background
- The Multi-processor ISA
- Matrix transpose example
- Conclusions

Globally Synchronous Load-Store Instructions

Program the memory, not the processor.

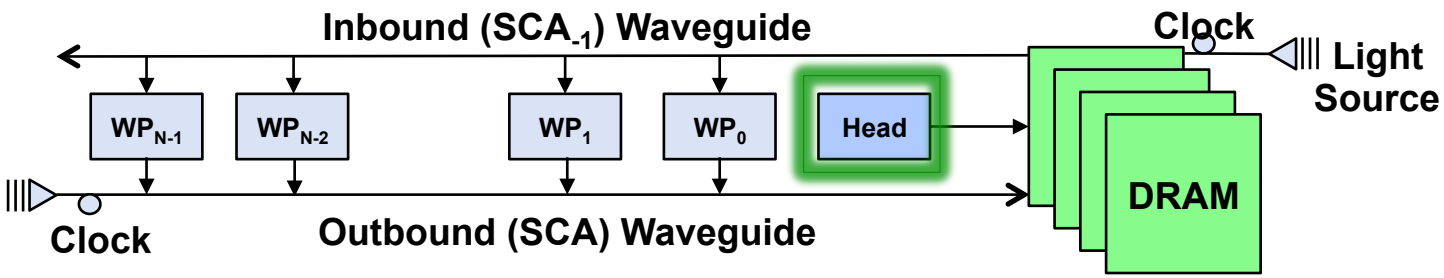
- **One coordinating instruction to initialize the communication schedule**

```
coalesce_sca        base_address, size
```

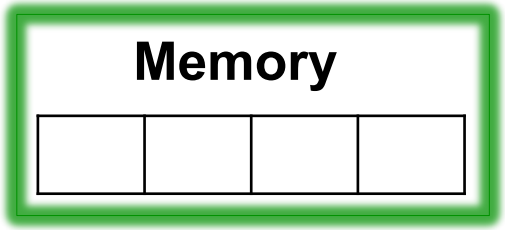
- **One instruction to write into the address space set up by the coordinating instruction.**

```
sca.b32             local_data, sca_index
```

Matrix Transpose on P-sync

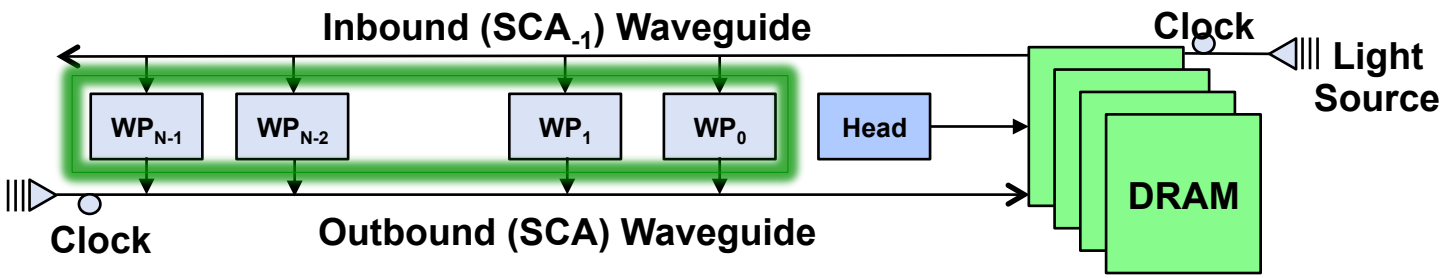


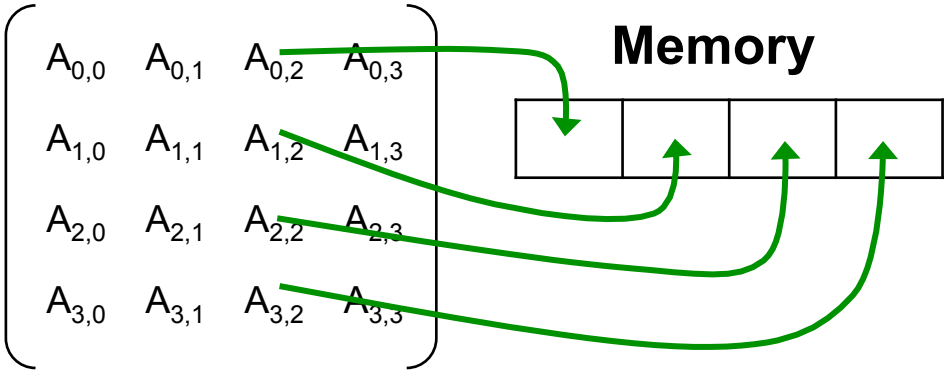
$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}$$



Head	`coalesce_sca Base, N`

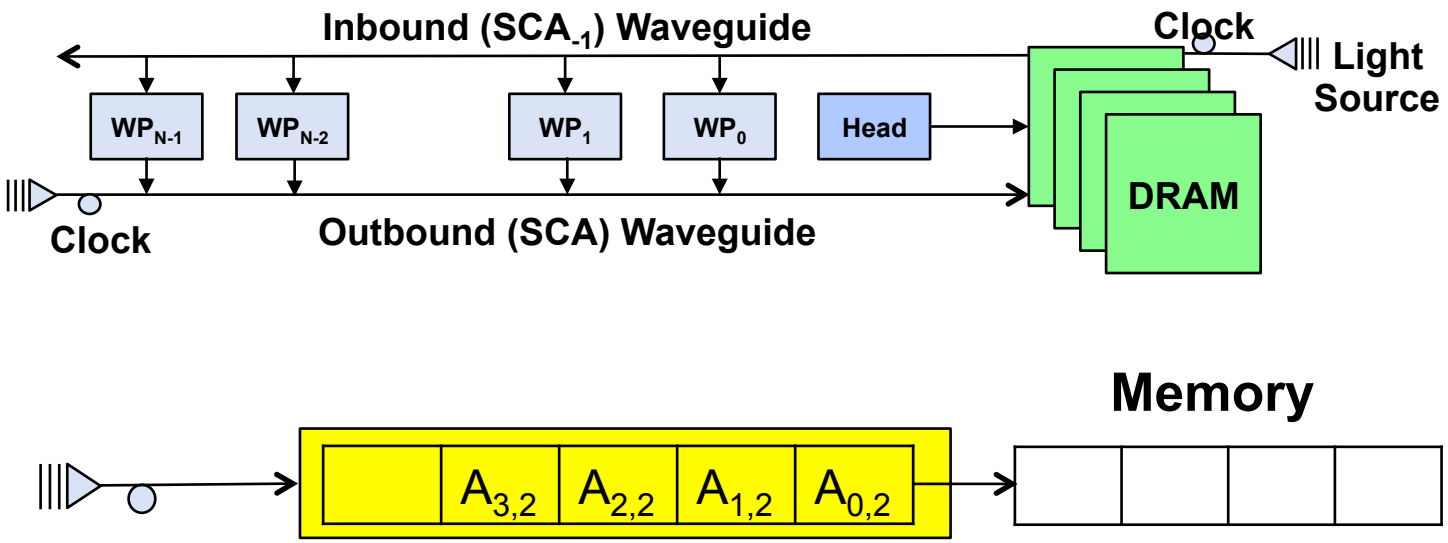
Matrix Transpose on P-sync





WP_0	`sca.b32 local[2], 0`
WP_1	`sca.b32 local[2], 1`
WP_2	`sca.b32 local[2], 2`
WP_3	`sca.b32 local[2], 3`

Matrix Transpose on P-sync



Distributed non-local data is combined on the waveguide to form a single efficient memory transaction.

Full Transpose Code

Head Node

```
R1 = global_base_address
R2 = 0          // current row
R3 = num_rows // loop count

.loop:
   // R4 = global address of row
   // to coalesce
   // R4 = R1 + (R2 * row_size)

   coalesce_sca R4, row_size

   add.32 R2, R2, 1 // current row
   sub.32 R3, R3, 1 // loop count

   // continue until all rows have
   // been coalesced.
bra loop
```

Worker Node

```
R1 = local_base_address
R2 = 0          // row index
R3 = row_size // loop count

.loop:

   // R5 = local data
   ld.local.u32 R5, local[R2]

   sca.b32 R5, proc_id

   add.u32 r2, r2, 1 // pos in row
   sub.u32 r1, r1, 1 // loop count

   // continue until this row has
   // been coalesced.
bra loop
```

Full Transpose Code

Head Node

```
R1 = global_base_address
R2 = 0          // current row
R3 = num_rows // loop count

.loop:
   // R4 = global_base_address of r
   // to coalesce
   // R4 = R1 + (R2 * row_size)

   coalesce_sca R4, row_size

   add.32 R2, R2, 1 // current row
   sub.32 R3, R3, 1 // loop count

   // continue until all rows have
   // been coalesced.
bra loop
```

- coalesce_sca executed once for each row of the matrix.
- Each time through the loop, it will wait for the prior coalesce to complete

Worker Node

```
   // R5 = local data
   ld.local.u32 R5, local[R2]

   sca.b32 R5, proc_id

   add.u32 r2, r2, 1 // pos in row
   sub.u32 r1, r1, 1 // loop count

   // continue until this row has
   // been coalesced.
bra loop
```

Full Transpose Code

Head Node

- Each processor has the same sca_index each time through the loop
- It is the head node that sets the global address
- The sca.b32 instruction will wait for its sca_index each time through the loop

```
    // R4 = R1 + (R2 * ro

    coalesce_sca R4, row_size

    add.32 R2, R2, 1 // current row
    sub.32 R3, R3, 1 // loop count

    // continue until all rows have
    // been coalesced.
bra loop
```

Worker Node

```
R1 = local_base_address
R2 = 0          // row index
R3 = row_size // loop count

.loop:

    // R5 = local data
    ld.local.u32 R5, local[R2]

    sca.b32 R5, proc_id

    add.u32 r2, r2, 1 // pos in row
    sub.u32 r1, r1, 1 // loop count

    // continue until this row has
    // been coalesced.
bra loop
```

Related

- General purpose CPUs
  - Optimized for locality (caches, prefetchers, etc.)
  - When there is no locality, these optimizations penalize performance
  - Existing cache-bypassing operations are single-thread
- GPUs
  - Coalescing only works within a Warp
  - The Kepler Shuffle Instruction manipulates data within a Warp

In these solutions the programmer cannot express global memory transactions across the entire architecture

Conclusions

- We introduce two new instructions that permit efficient multi-processor synchronization.
  - coalesce_sca
  - sca
- These new instructions, in combination with SCA capability, give us
  - simple code
  - high network and memory efficiency due to well-coordinated communication

Global synchrony enables parallel efficiency