The Sparse Data Reduction Engine

Chopping Sparse Data one byte at a time

Jonathan Beard

Staff Research Engineer, Arm Inc.

MEMSYS 2017

Disclaimer

This is a research project, not product.

Any changes to architecture must go through the standard architecture review board process

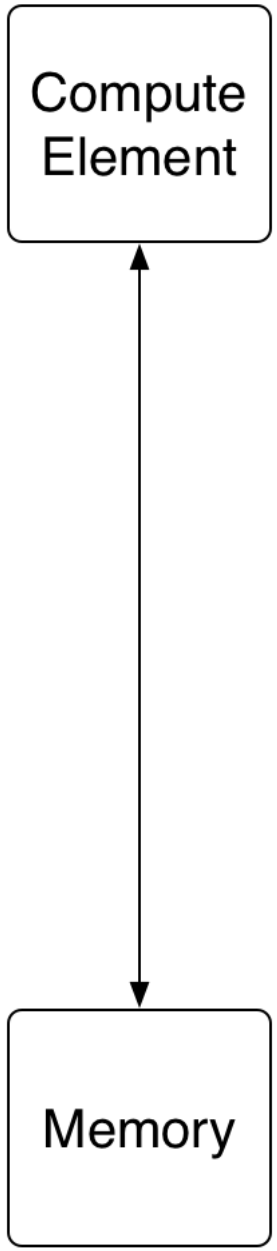
Nothing within this presentation should be taken as a direction of future architecture or product plans for Arm Inc/Ltd.

Please ask questions related to the presentation!

Data movement dominates in the post-Moore era

- Well understood problem, computing typically takes less energy than moving the data (~2-80x).
- Solutions? More exotic technologies?
- Next 5-10 years?
  - Building up and out: 3D stacked memories and 3D stacked compute (maybe, once we have better design tools)
  - Non-volatiles integrated and slowly replacing standard volatile memory (due to lack of refresh)
  - Short term: an architecture explosion….

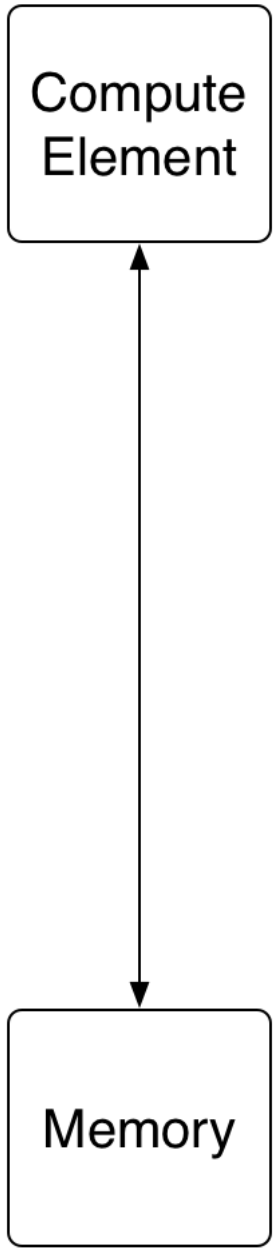
(Florence, Italy)

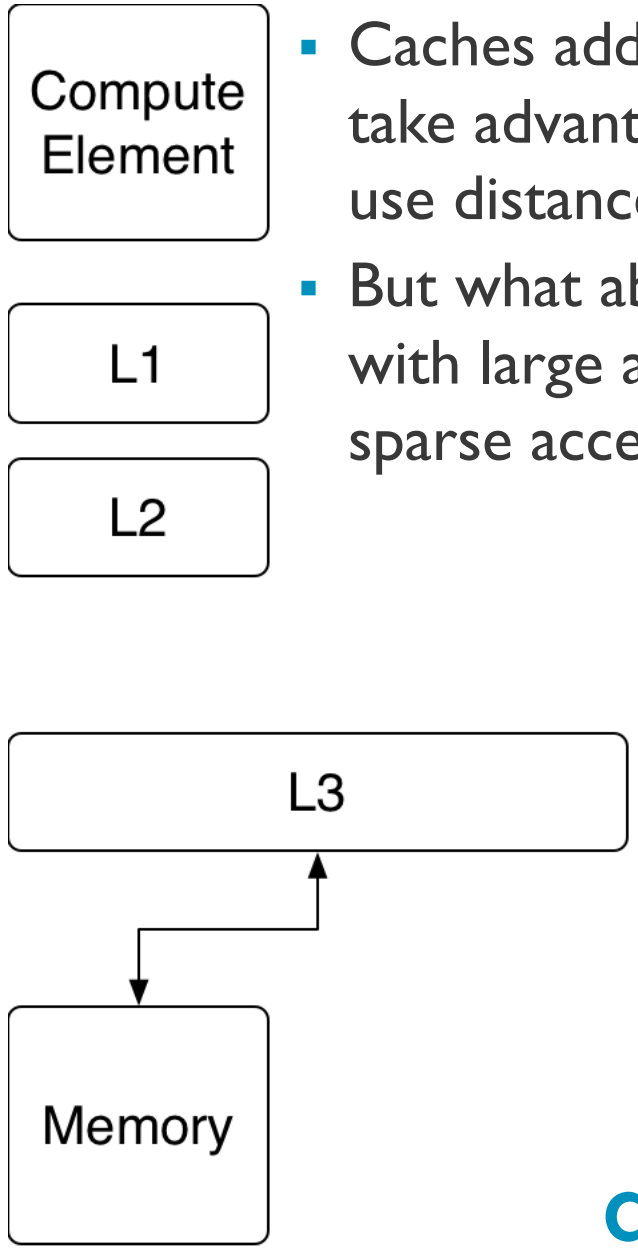


Basic Von Neumann architecture

A compute element

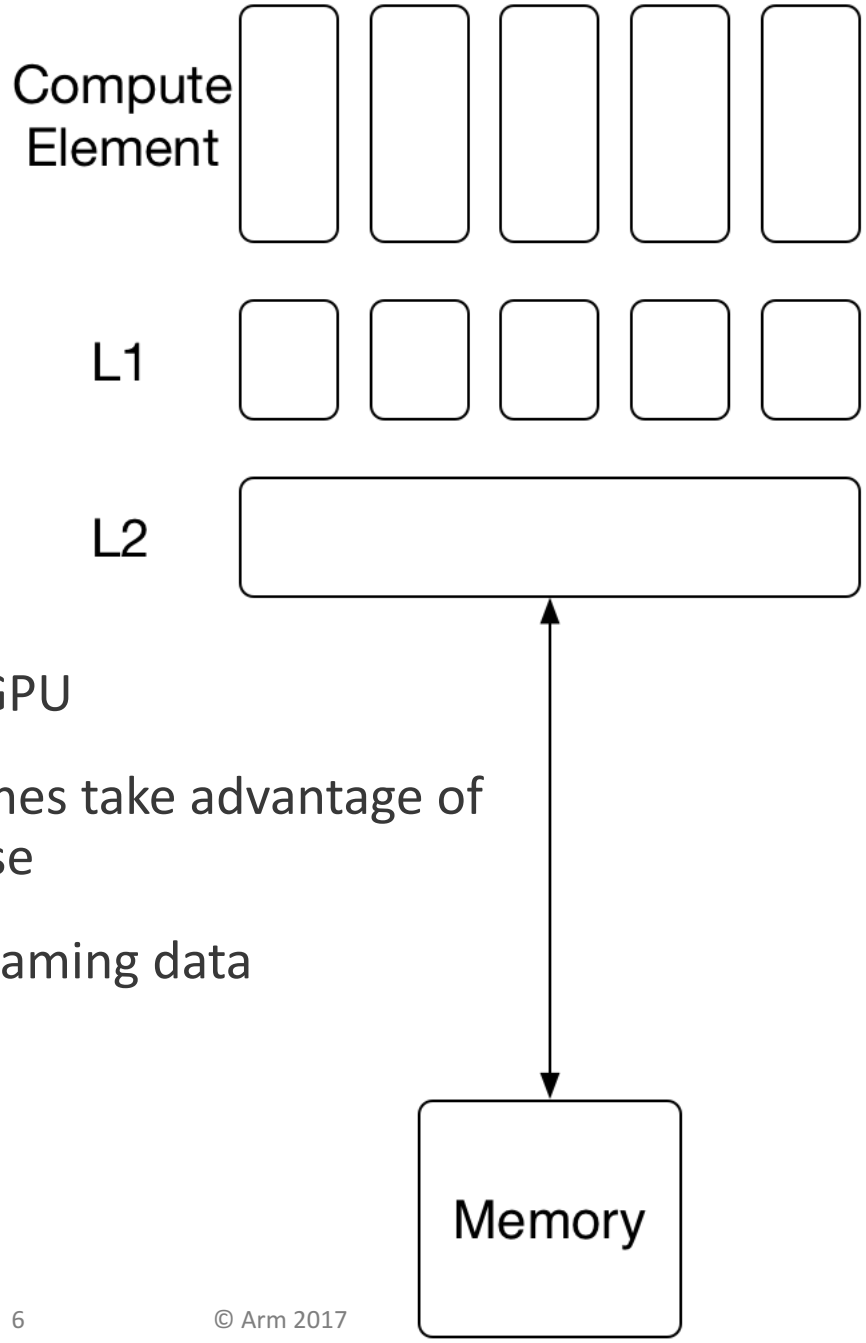
A memory interface





- Caches added to take advantage of re-use distance
- But what about apps with large and/or sparse accesses?

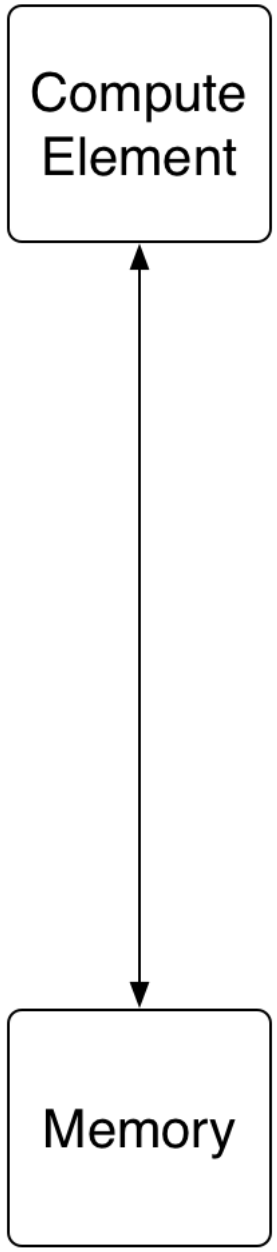
arm

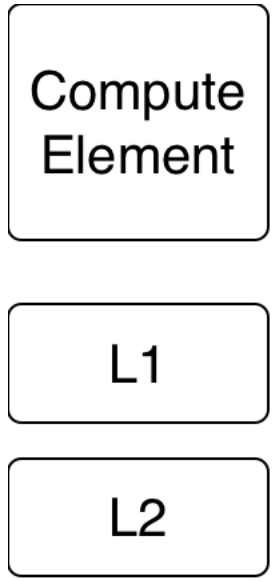


GPGPU

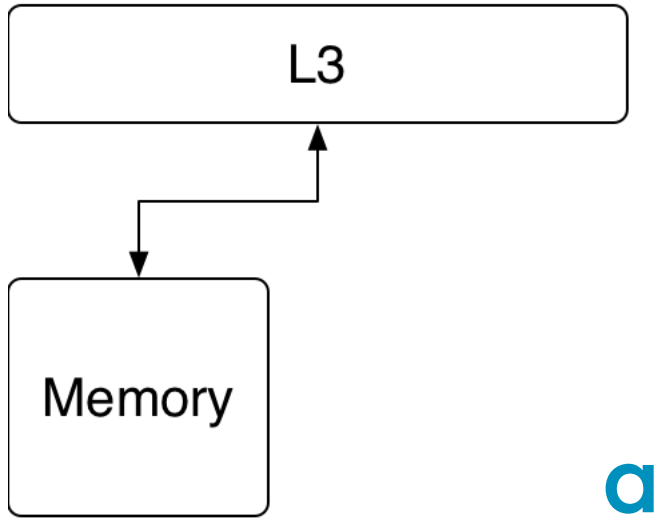
Caches take advantage of reuse

Streaming data

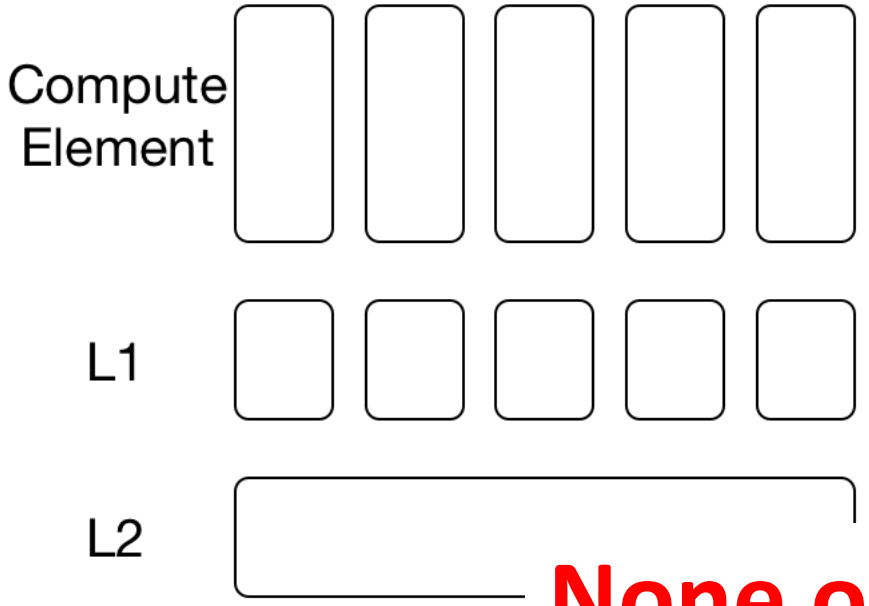


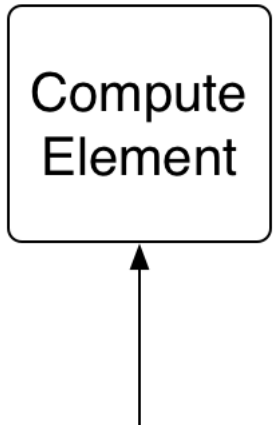


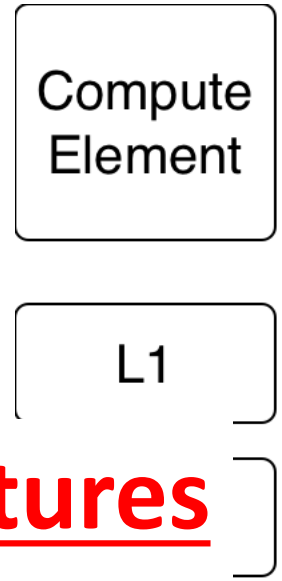
- Caches added to take advantage of re-use distance
- But what about apps with large and/or sparse accesses?



arm







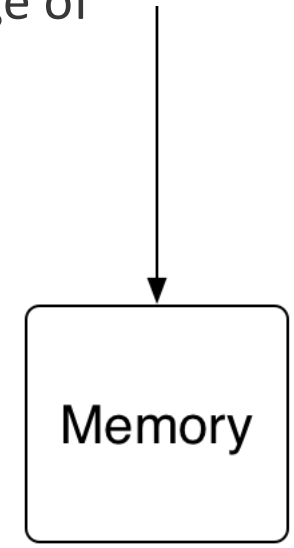
- Caches added to take advantage of re-use distance
- But what about apps with large and/or sparse accesses?

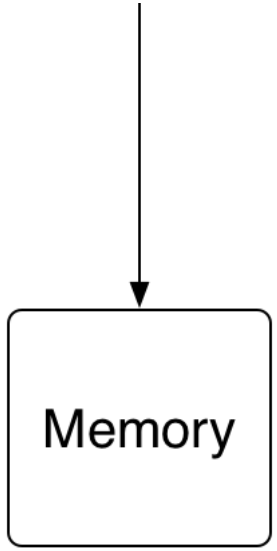
None of these architectures handles sparse data well!

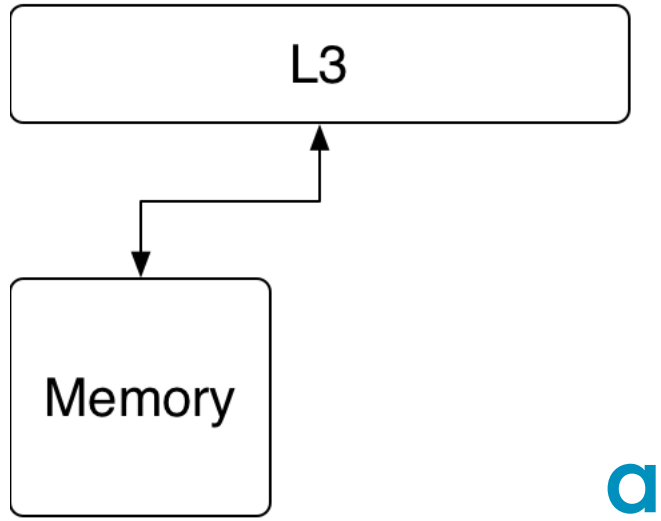
GPGPU

Caches take advantage of reuse

Streaming data

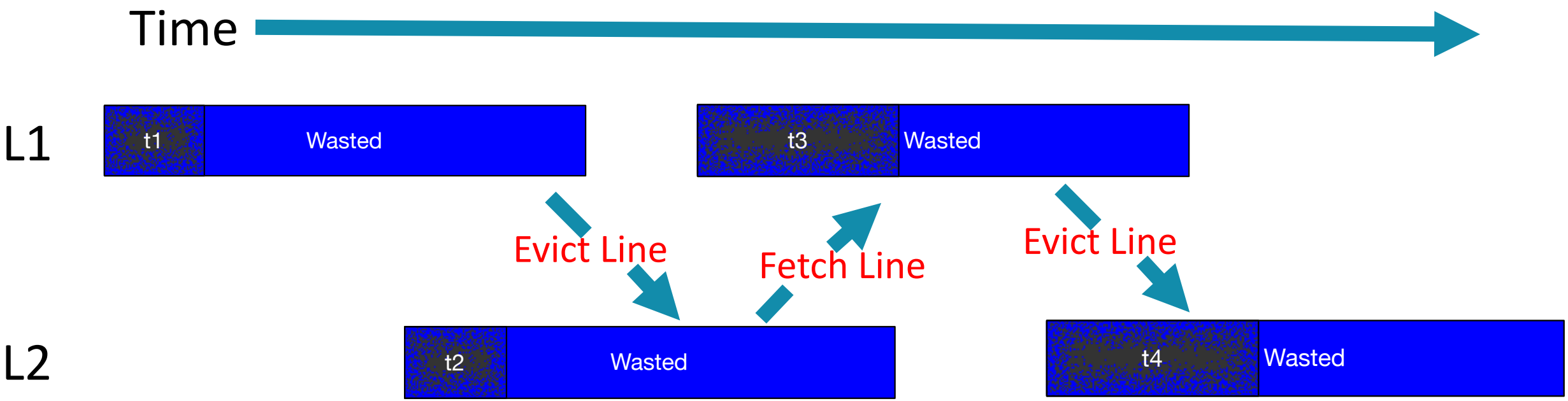






arm

Dance of reuse and utilization



arm

Superfluous data movement

Utilized Bandwidth

Wasted Bandwidth

arm

Dark Bandwidth:

Data moved throughout the memory hierarchy that is not used after it is moved.

Dark Bandwidth:

Unused data moved throughout the memory hierarchy not at request of programmer but because of hardware design.

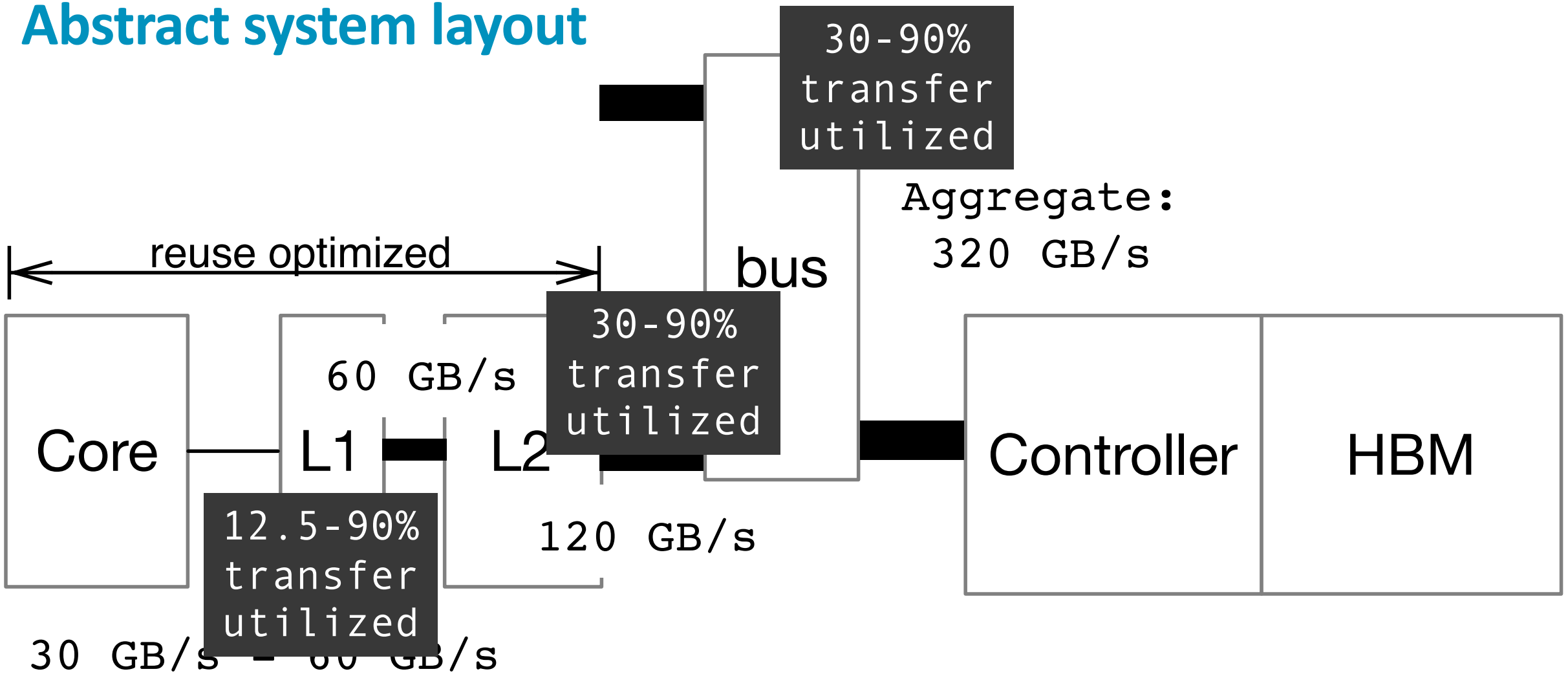
Superfluous data movement

Utilized Bandwidth

Dark Bandwidth

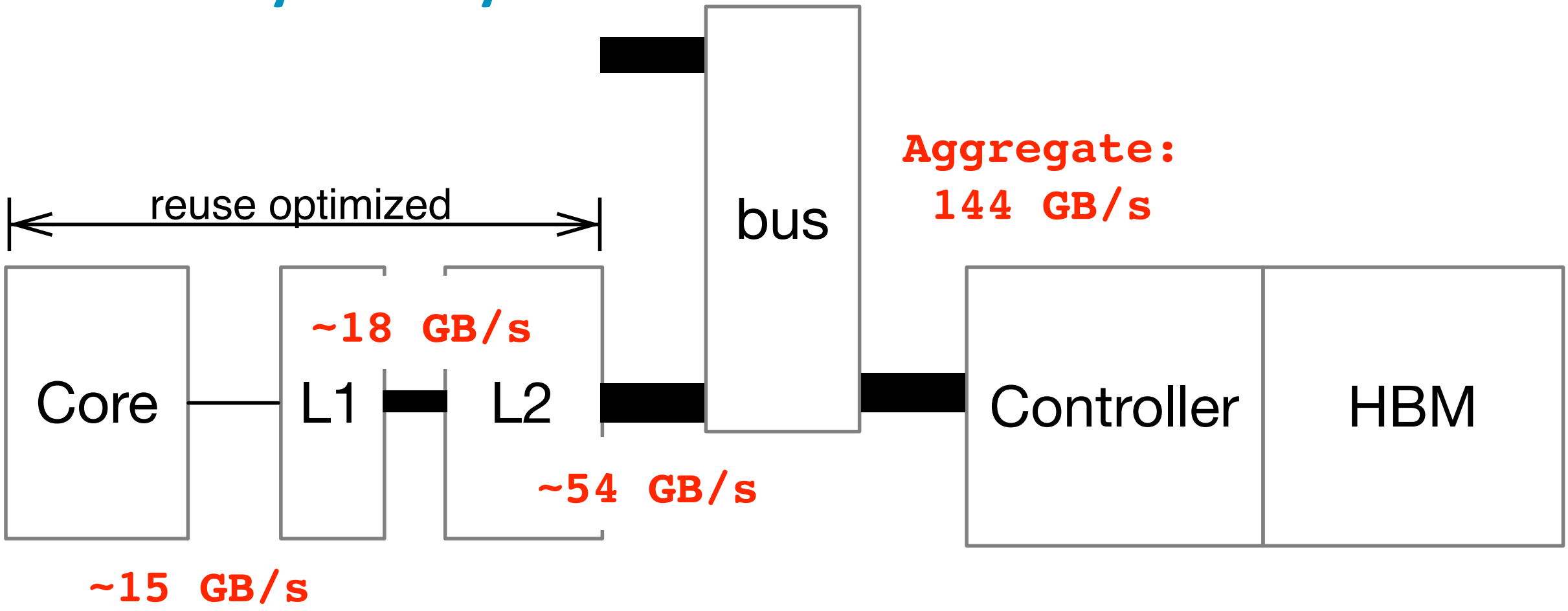
arm

Abstract system layout



arm

Abstract system layout

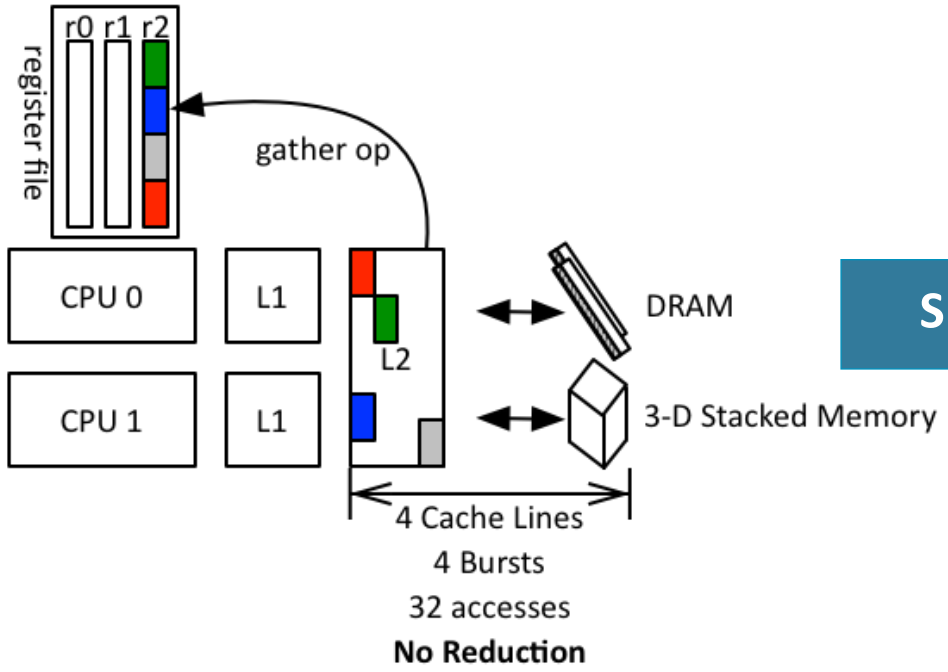


arm

Compacting Data
First: The Alternatives

Sparse Data Reduction (mechanisms)

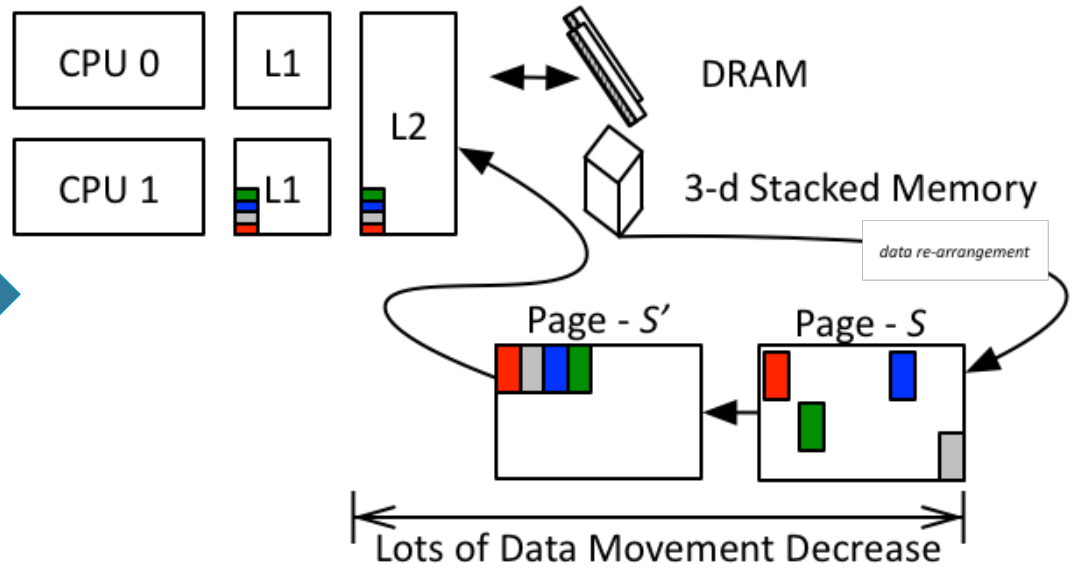
Standard Gather-Scatter (In Core)



- Smaller binary (e.g., single gather vs. multiple loads)

SPDRE

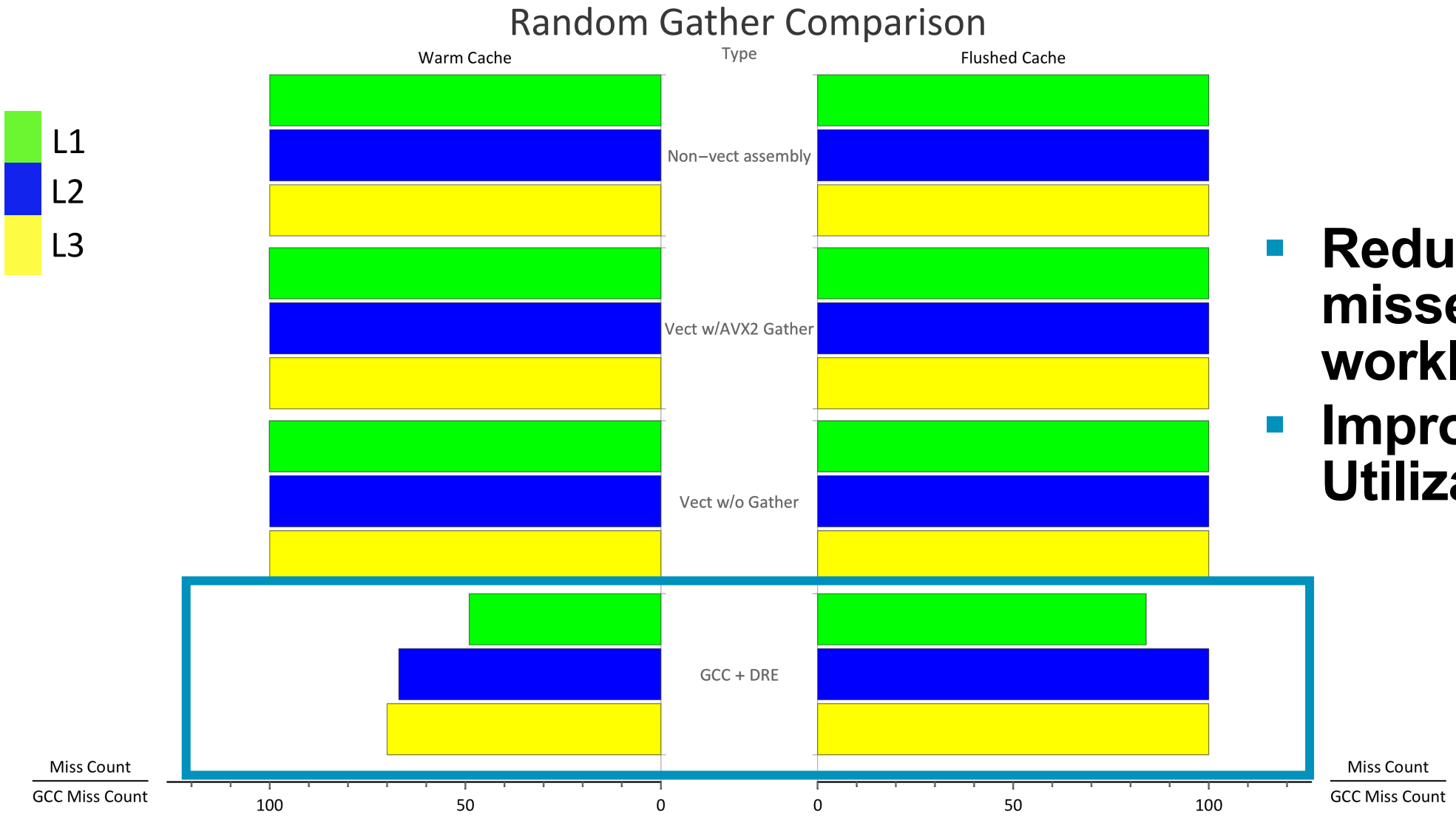
In-Memory Gather-Scatter



- Smaller binary (e.g., single gather vs. multiple loads)
- Improved Cache Line Utilization
- Reduced Overall Data Movement

arm

Improving on scatter/gather



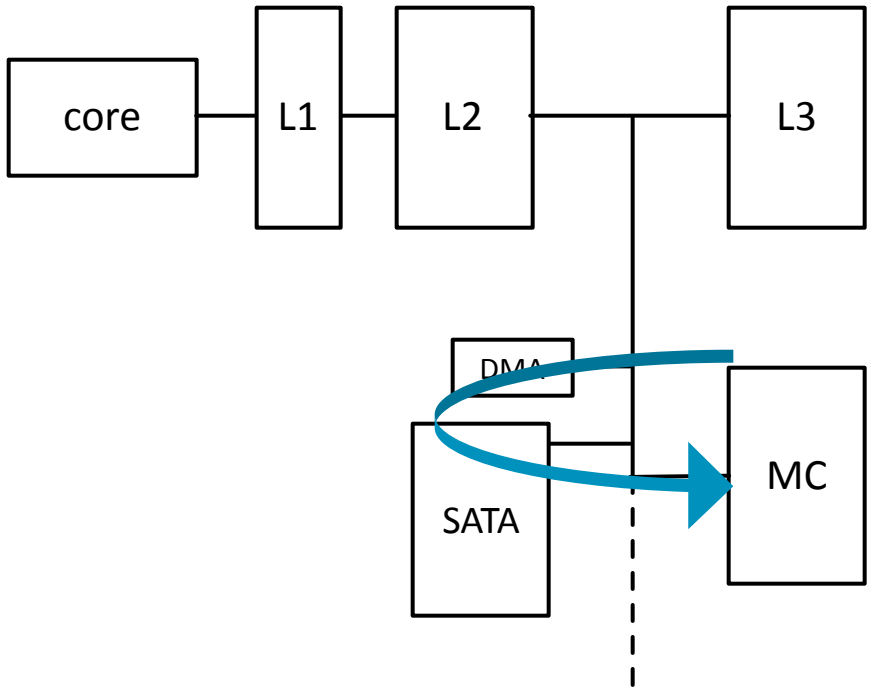
- **Reduce cache misses for sparse workloads**
- **Improve Bandwidth Utilization**

arm

Programmable DMA

- Generally limited to page gather/move/scatter
- Usually coherent (lots of transactions), wastes bus utilization (i.e., all traffic goes through coherent network)
- Often limited when it comes to finer granularity
- Typically no persistent state
- Not solving quite the same problem!

Toy Example:



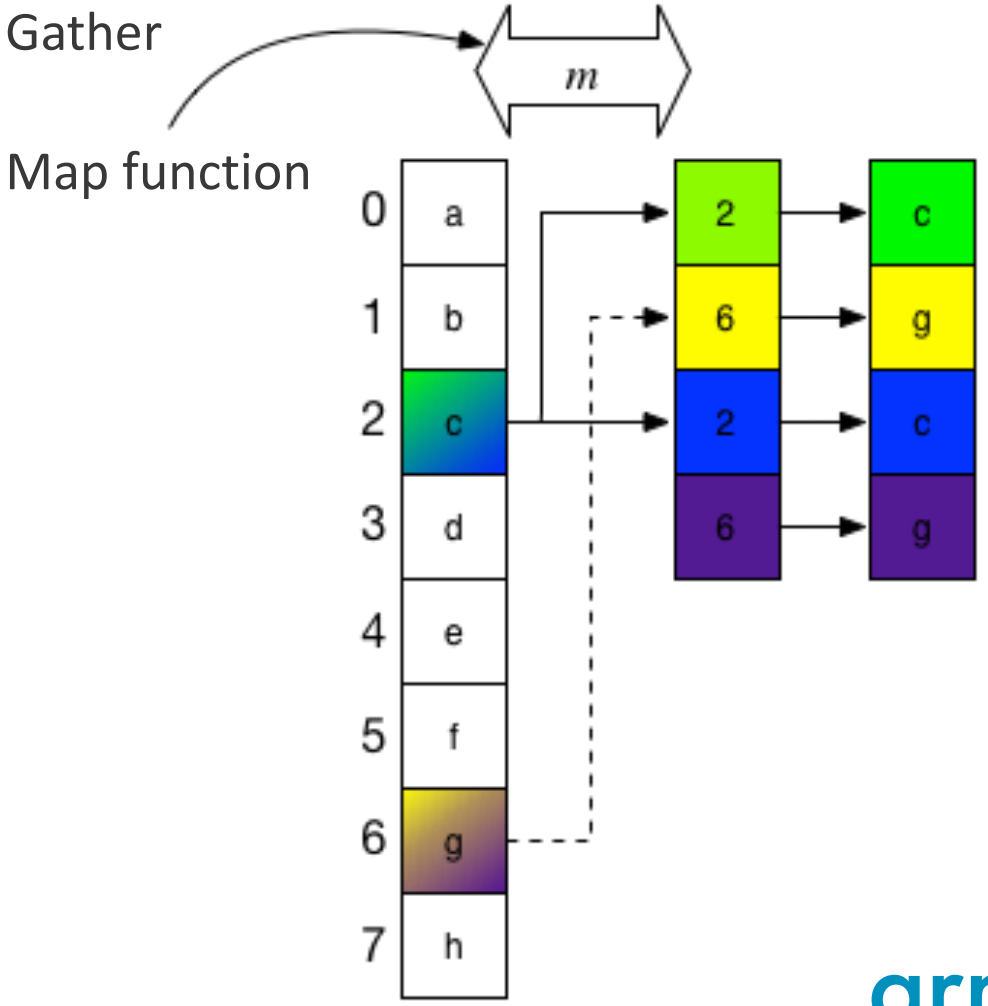
The SPiDRE

SPiDRE in words

Decoupling access and execute

The idea

- Programmers know what data they want (usually)
- Finding Compressed Sparse Row manipulations for new problems often takes a lot of time (bad for productivity)
- Why not build an interface that allows gathering (potentially) in any device of only the data you (the programmer) needs
- “Bulk” byte-addressed memory
- Doesn’t break coherence…with some caveats
- Can create multiple windows of original segment



arm

Programming SPiDRE (one approach)

```
template < typename TYPE >
   static void alloc( TYPE **output,
                      const std::size_t nitems );
```

User inserted allocate (new virtual and physical space)

User inserted rearrange function

```
template < class DST, class SRC >
   static std::size_t rearrange(
        DST * const dst,
        SRC * const src,
        const std::size_t nitems,
        rearrange_func_t< DST, SRC > src_dst );
```

```
template < class S, class S_Prime >
   static void sync( S       * const s,
                     S_Prime * const s_p,
                     const std::size_t nitems );
```

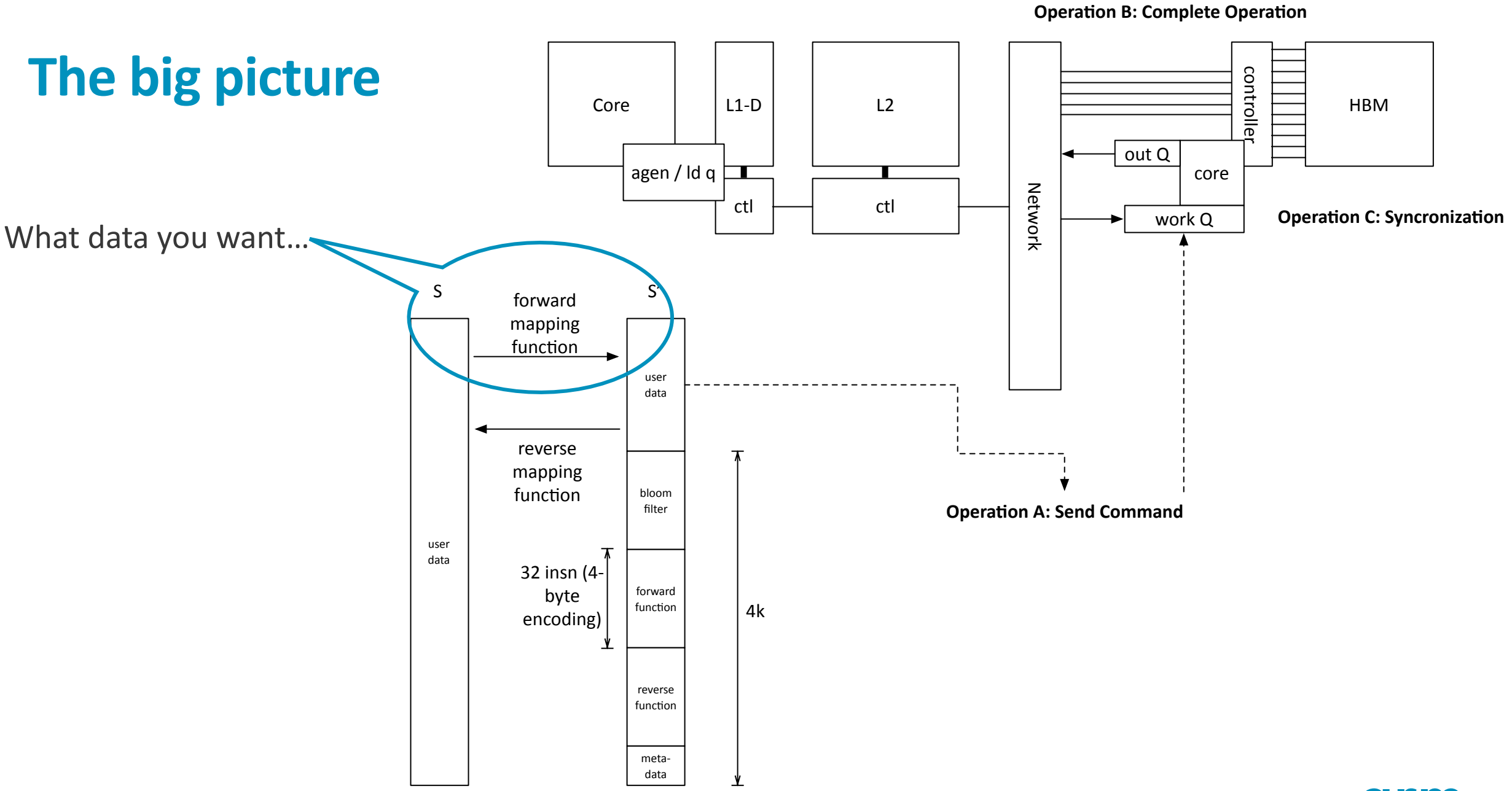
Synchronize S' -> S

```
template < typename TYPE >
   static void free( TYPE *ptr );
```

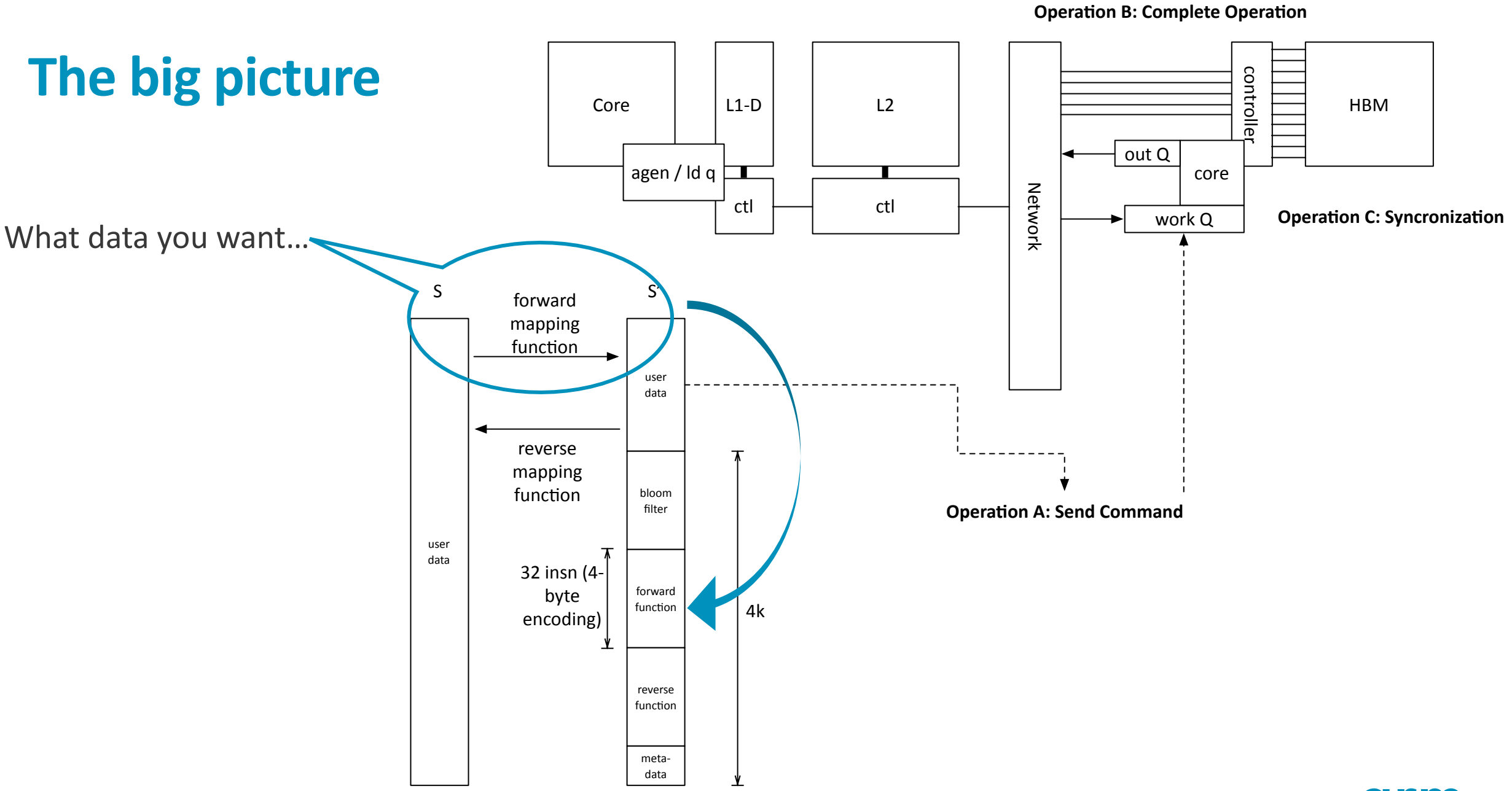
free

arm

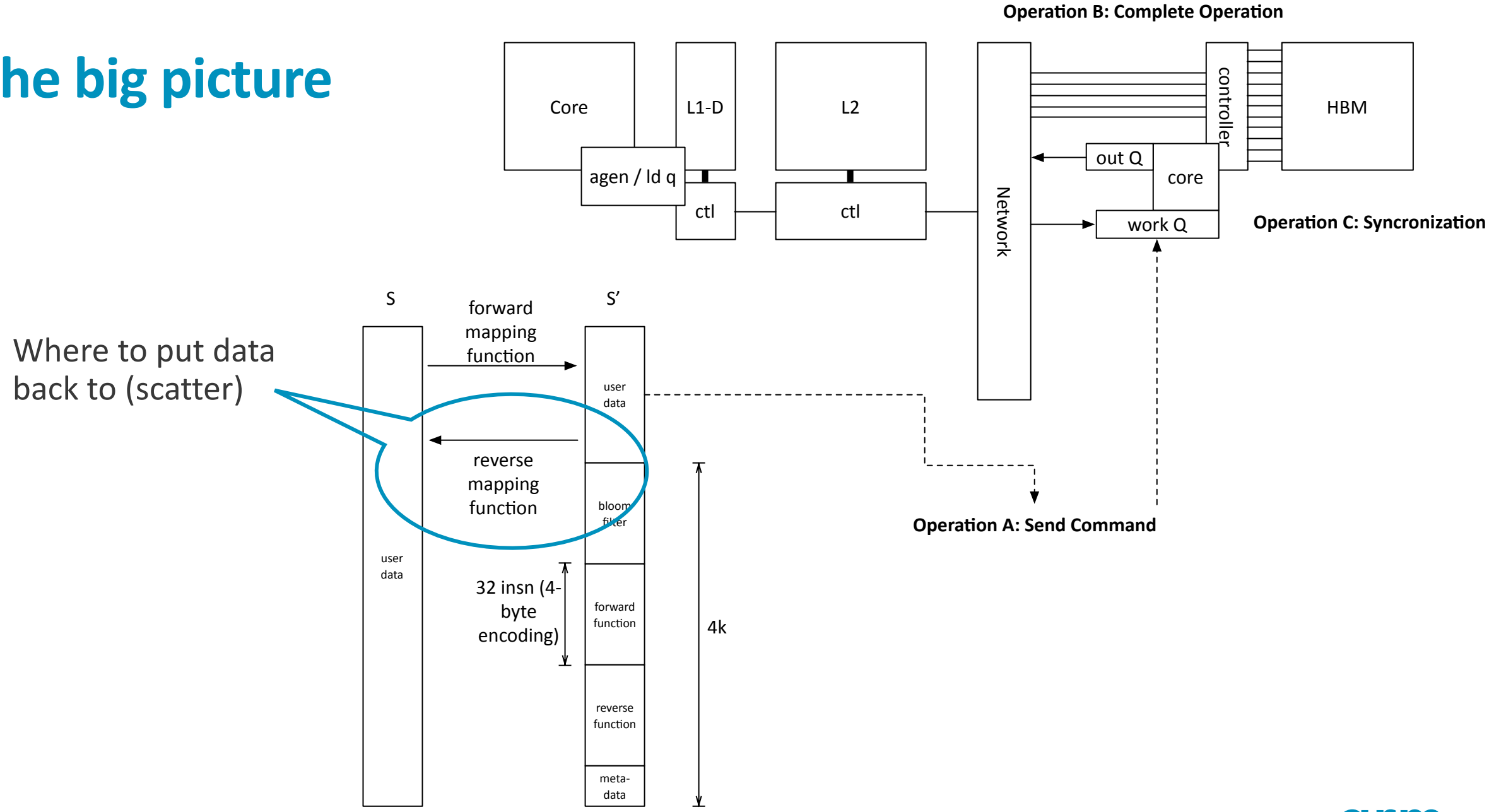
The big picture



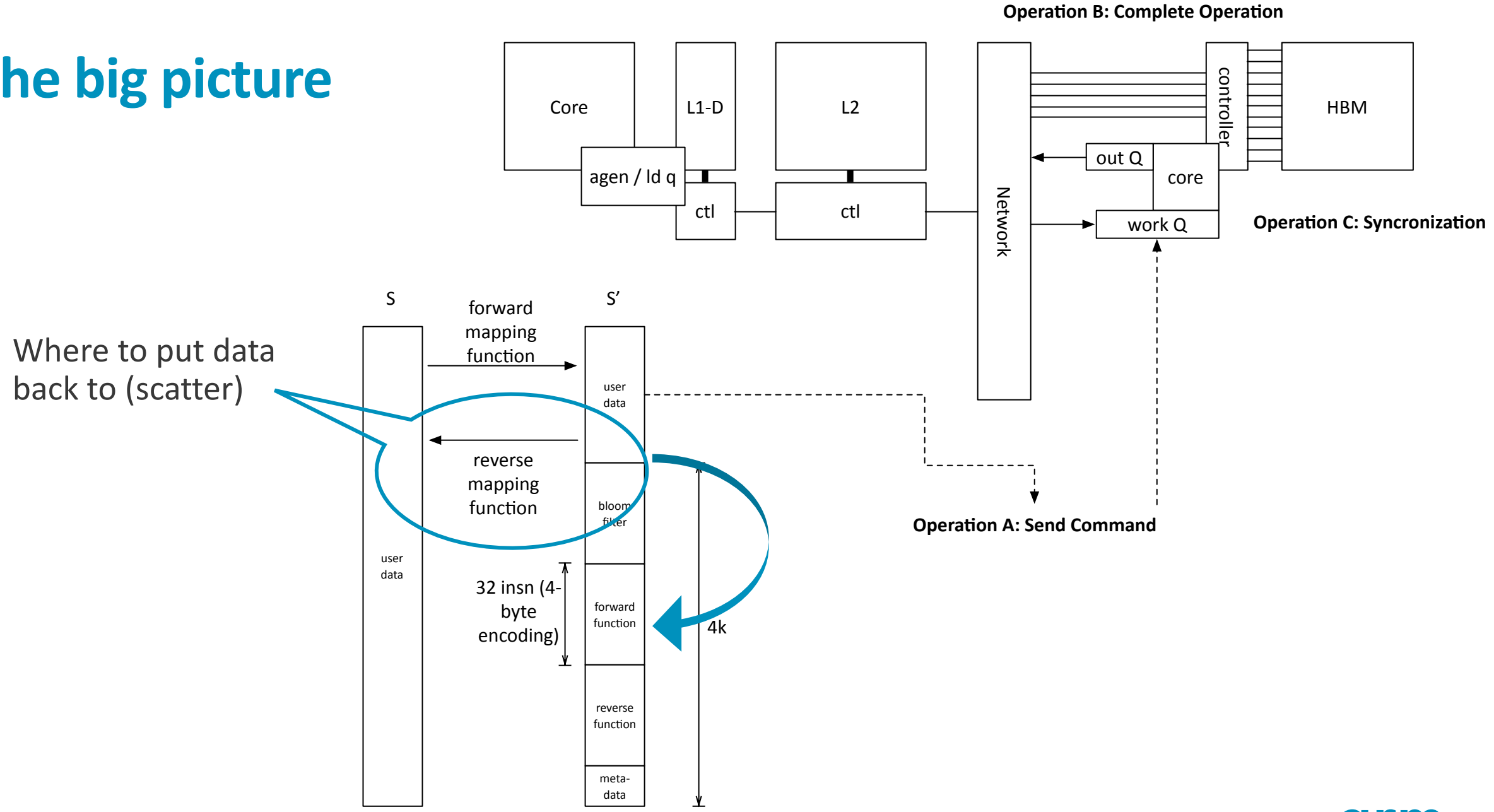
The big picture



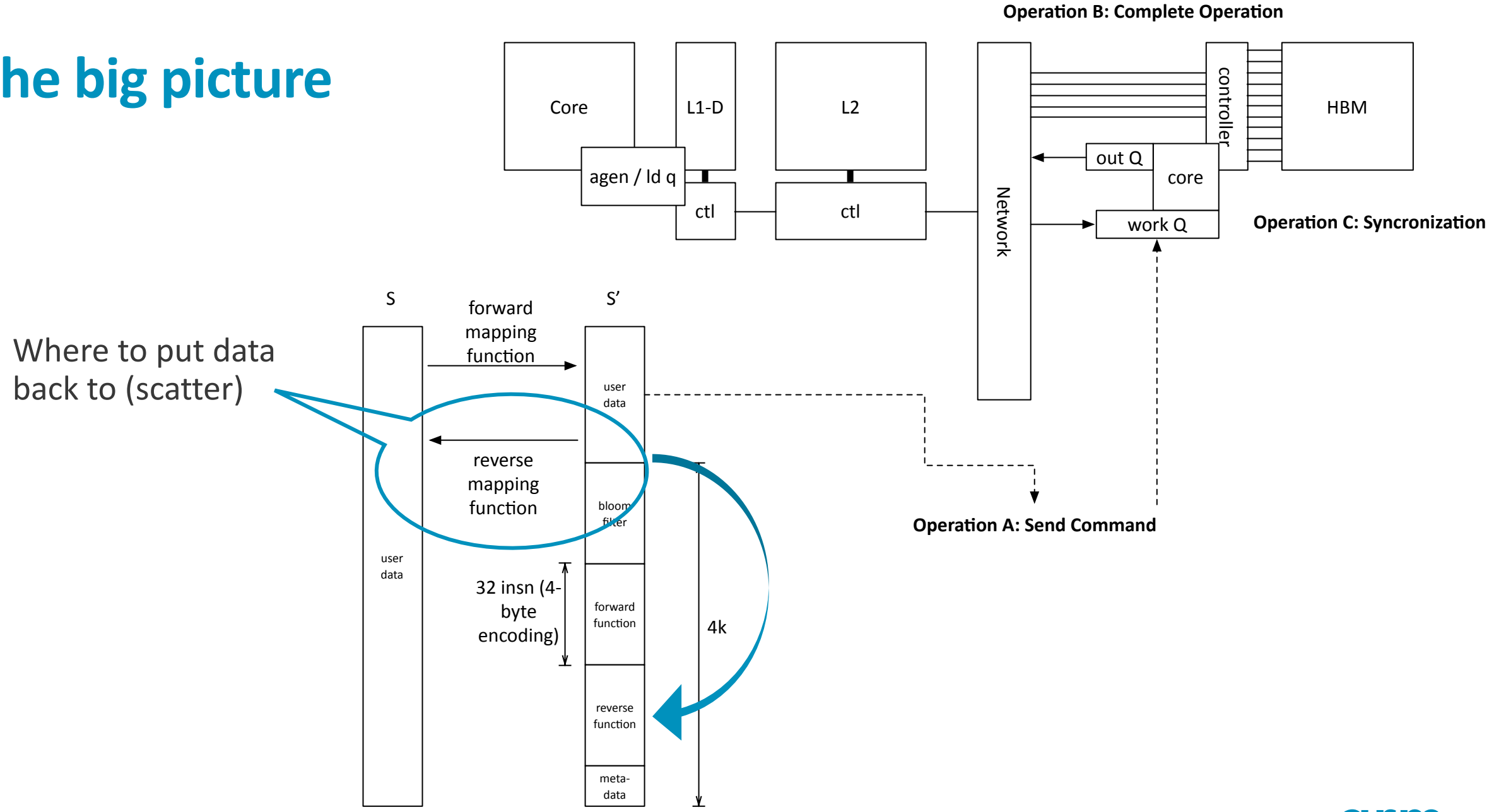
The big picture



The big picture

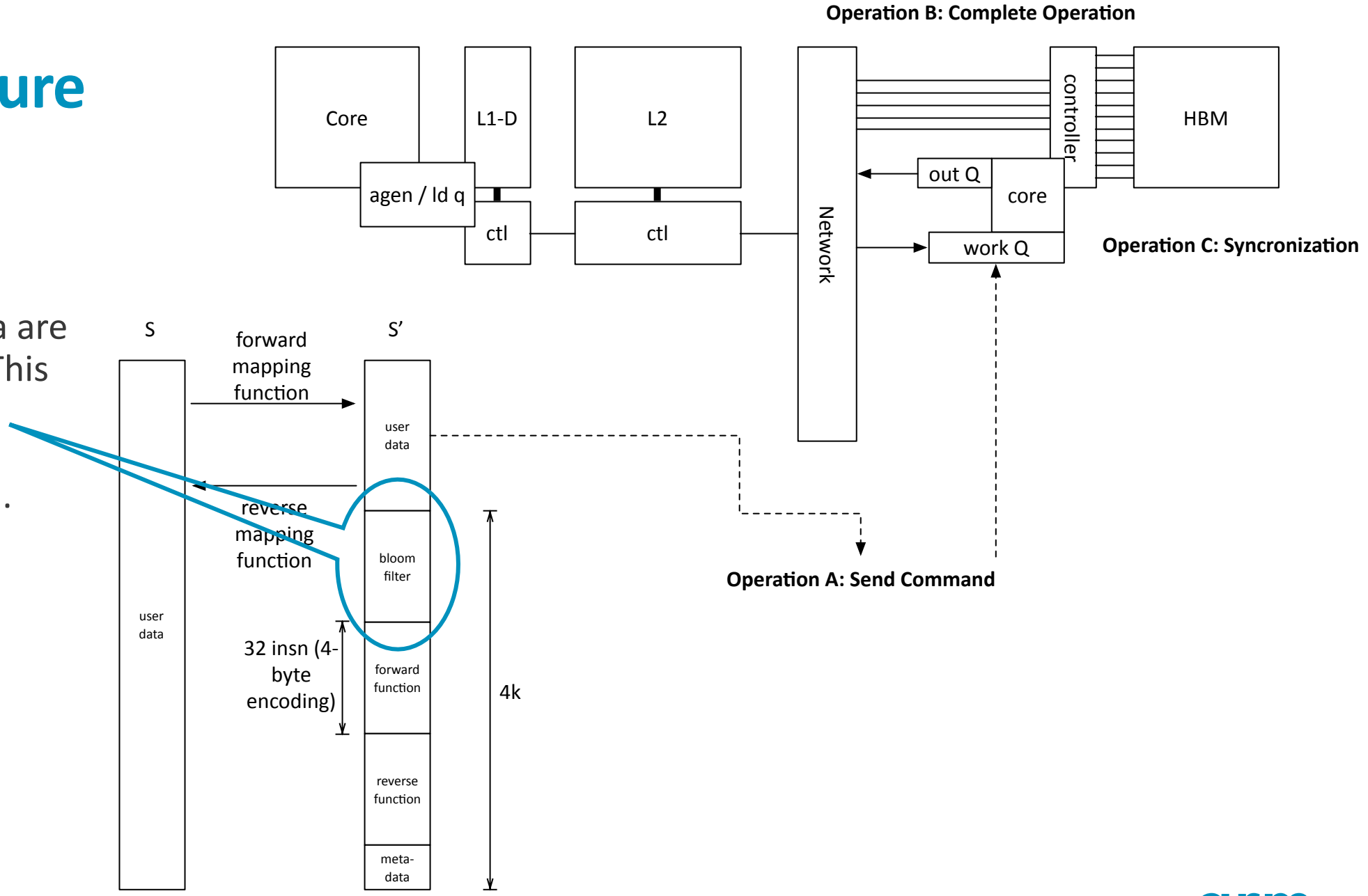


The big picture



The big picture

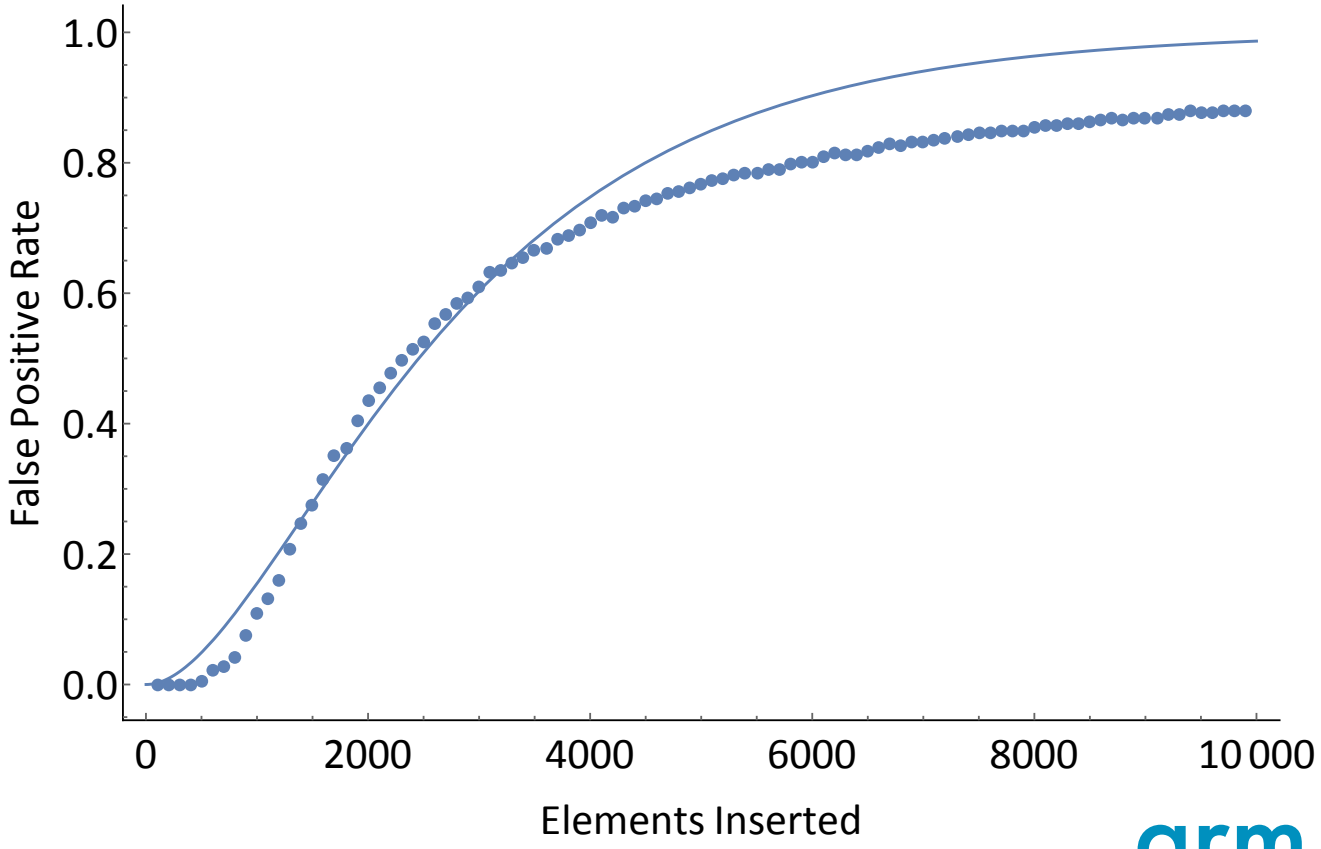
Save which data are modified in S'. This enables more efficient synchronization.



Bloom filter analysis

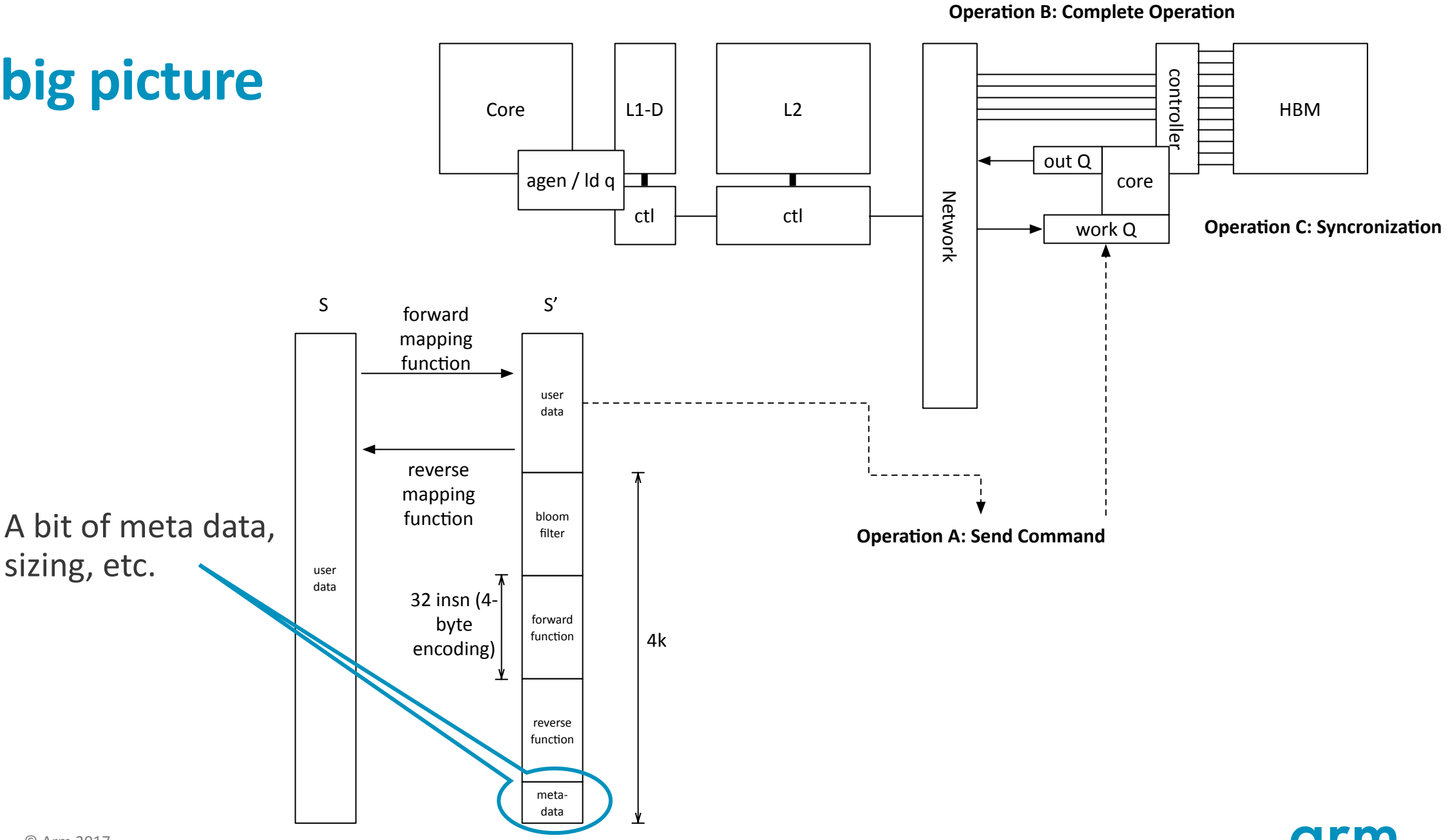
Quick and simple

- [image to right] dots are our implementation, line is theoretical
- With 4k bits, ~22% false positive rate, ~1500 elements, 78% reduction in unnecessary write-backs at *N* granularity (e.g., 64-byte, 4K, 64K, 2M)
- Method enables variable granularity set at allocation
- Bottom line: it's a well understood Bloom filter, enables reduction of write-back on synchronization

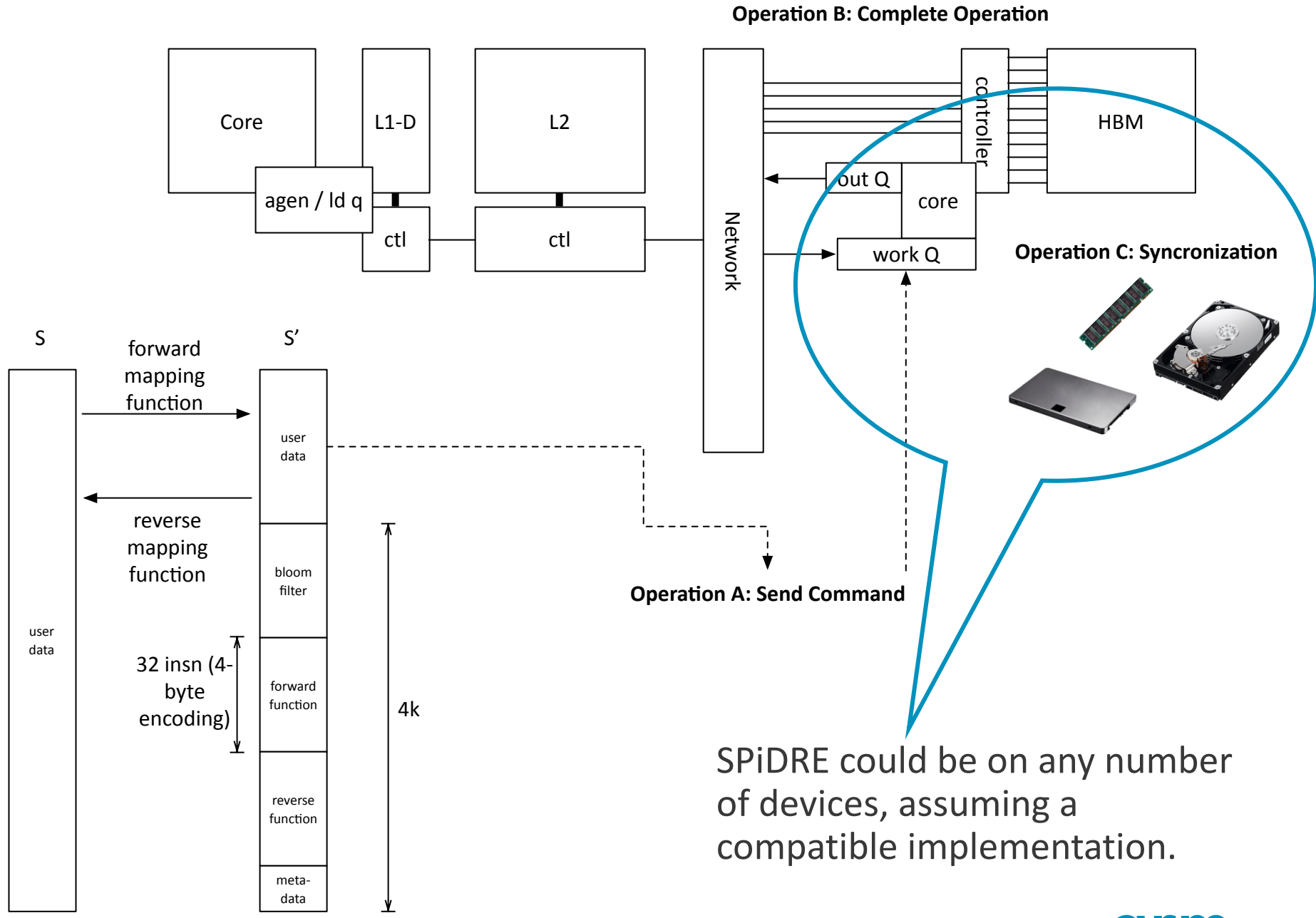


arm

The big picture



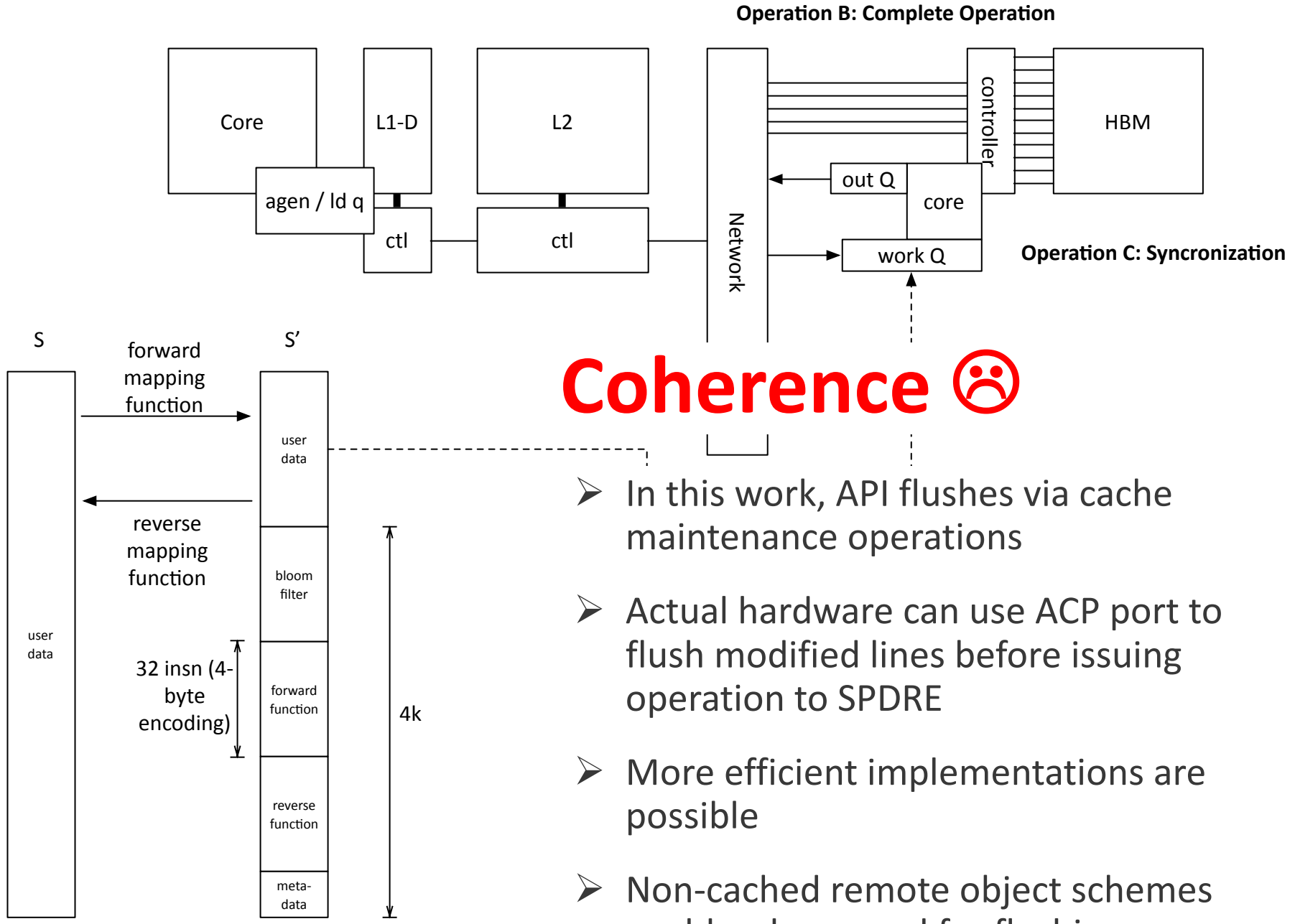
The big picture



SPiDRE could be on any number of devices, assuming a compatible implementation.

The big picture

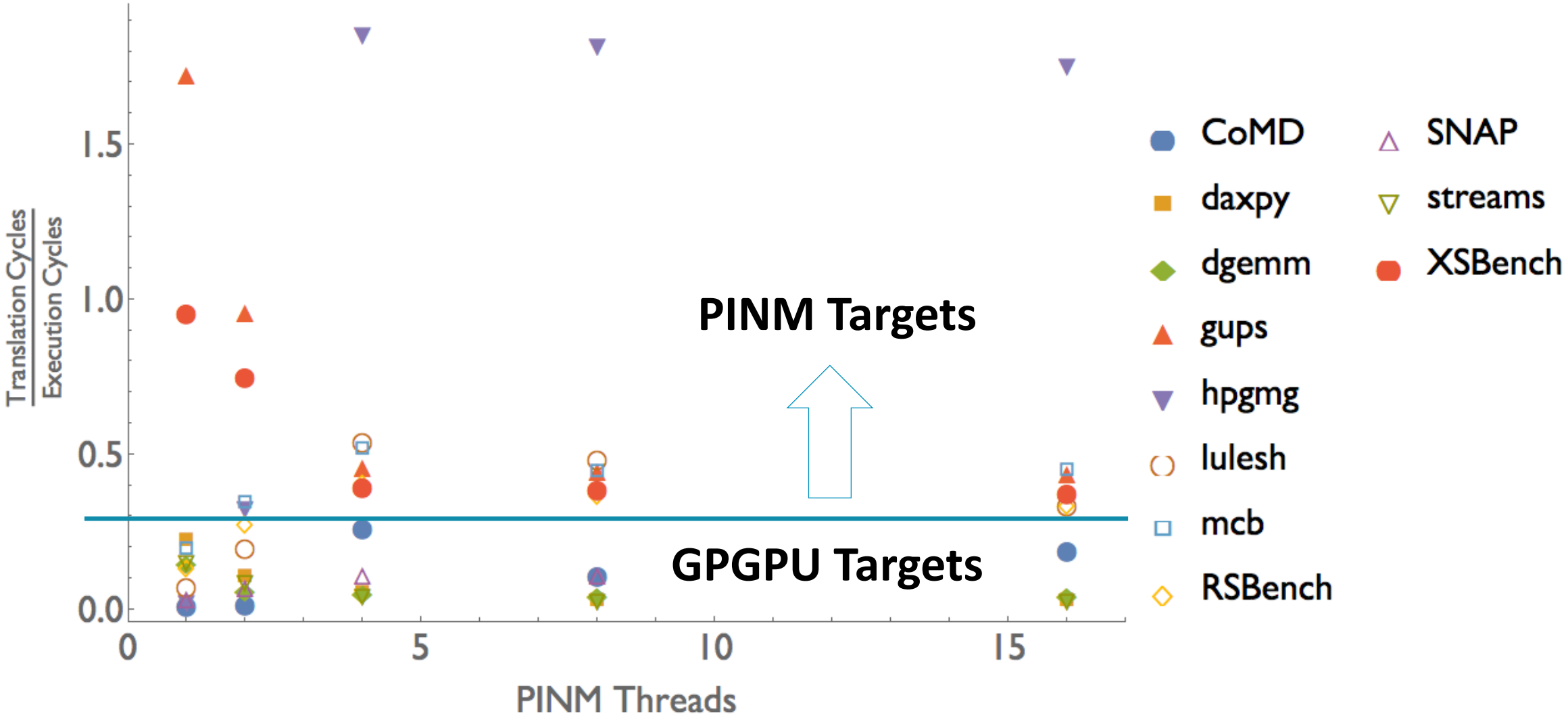
S marked as read only while **S'** exists



Coherence ☹

- In this work, API flushes via cache maintenance operations
- Actual hardware can use ACP port to flush modified lines before issuing operation to SPDRE
- More efficient implementations are possible
- Non-cached remote object schemes could reduce need for flushing

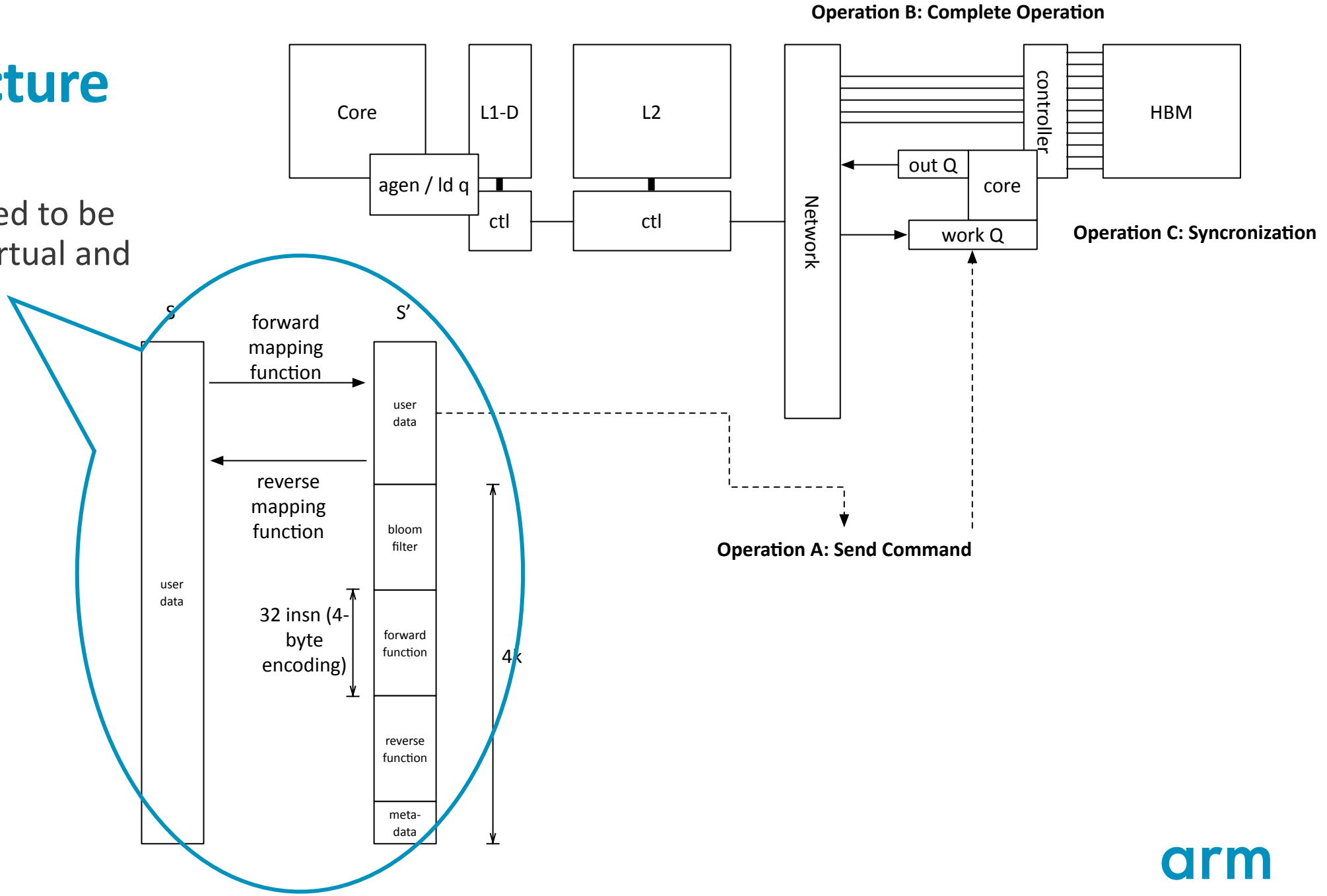
Sparse translation through IOMMU



arm

The big picture

Must be allocated to be contiguous in virtual and physical space…



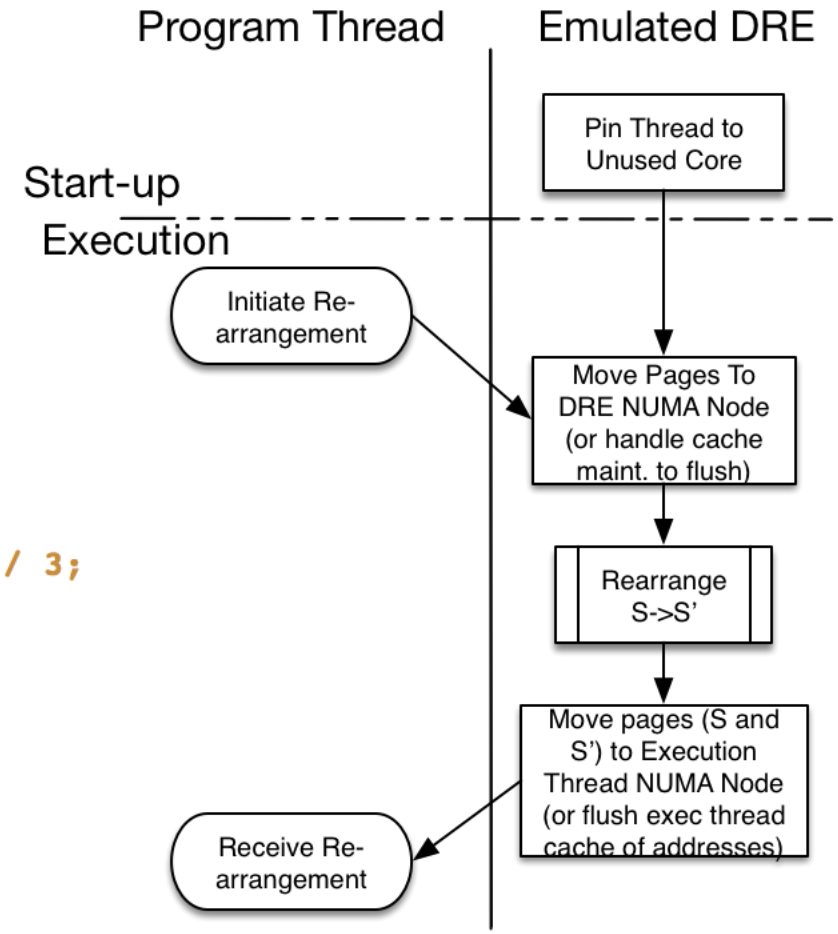
Sim Setup & Results

Simulation Environment (SPiDRE)

```
/** allocate DRE memory **/
dre::alloc( dst, source_length );
const auto num_rearranged(  dre::rearrange( *dst, src, source_length, offset ) );
for( auto dst_index( 0 ); dst_index < ret_val; dst_index++ )
{
  (*dst)[ dst_index ] = workload( (*dst)[dst_index] );
}
/** synchronize  **/
dre::sync( *dst, src, N_SRC );
dre::free( *dst );
dre::release( src, N_SRC );
```

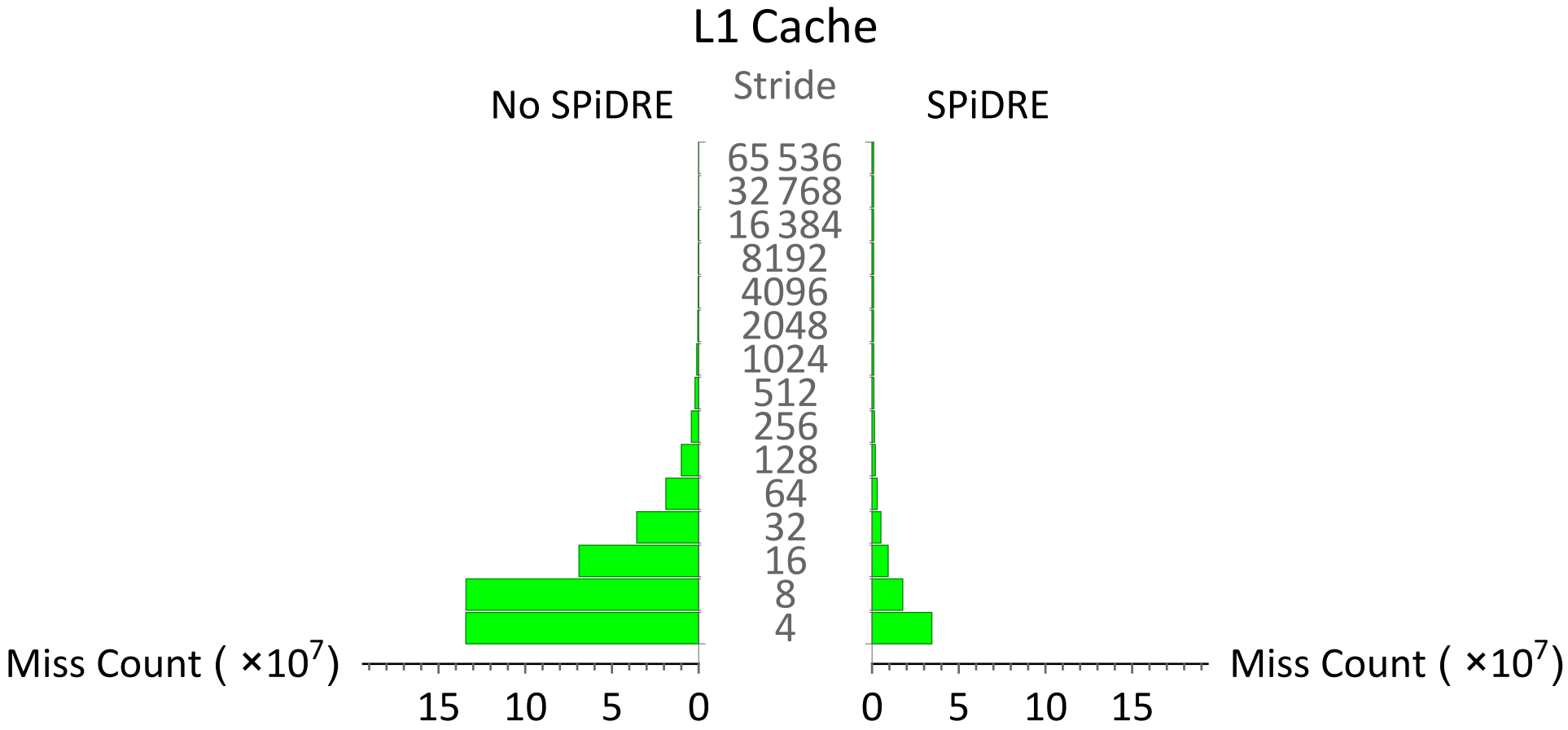
```
#define workload( x )\
   std::exp( static_cast< double>( x )  / 2 ) * 7.0f / 3;
```

Platform	Specifications
Intel	• 2x CPU E5-2690 v3 • 64 GB DDR4 PC4-17000 • Linux Kernel Version 3.13



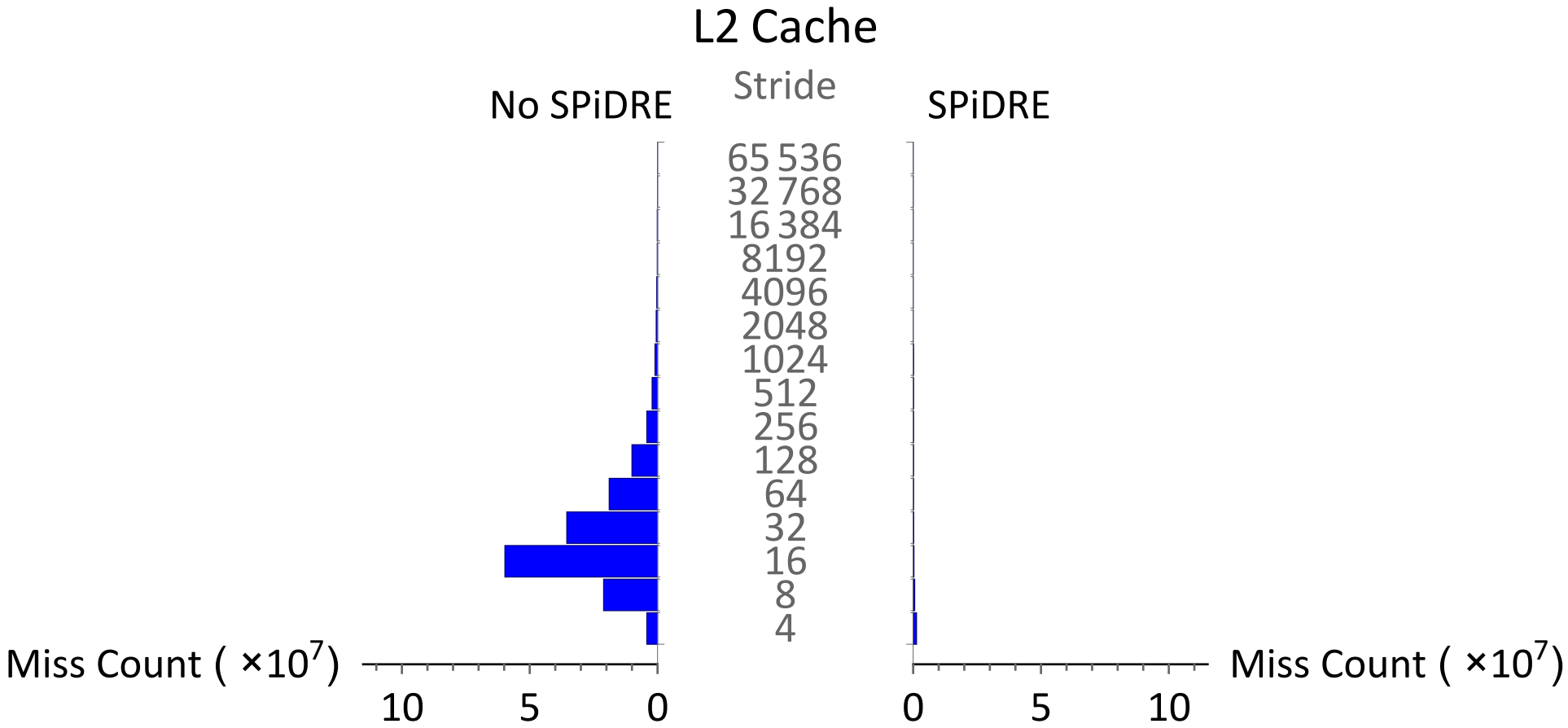
arm

Fixed Stride (1GB data set) - Gather SPiDRE



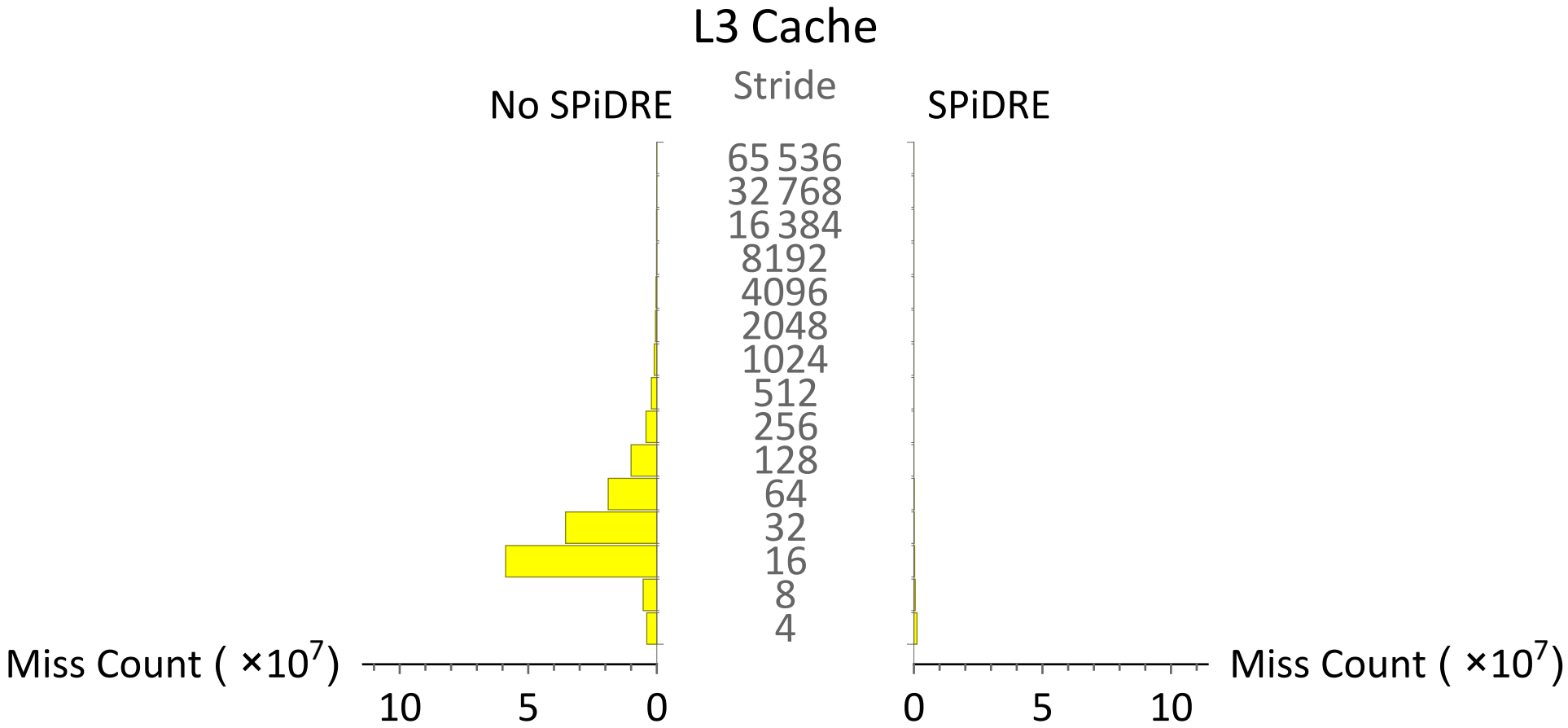
arm

Fixed Stride (1GB data set) - Gather SPiDRE



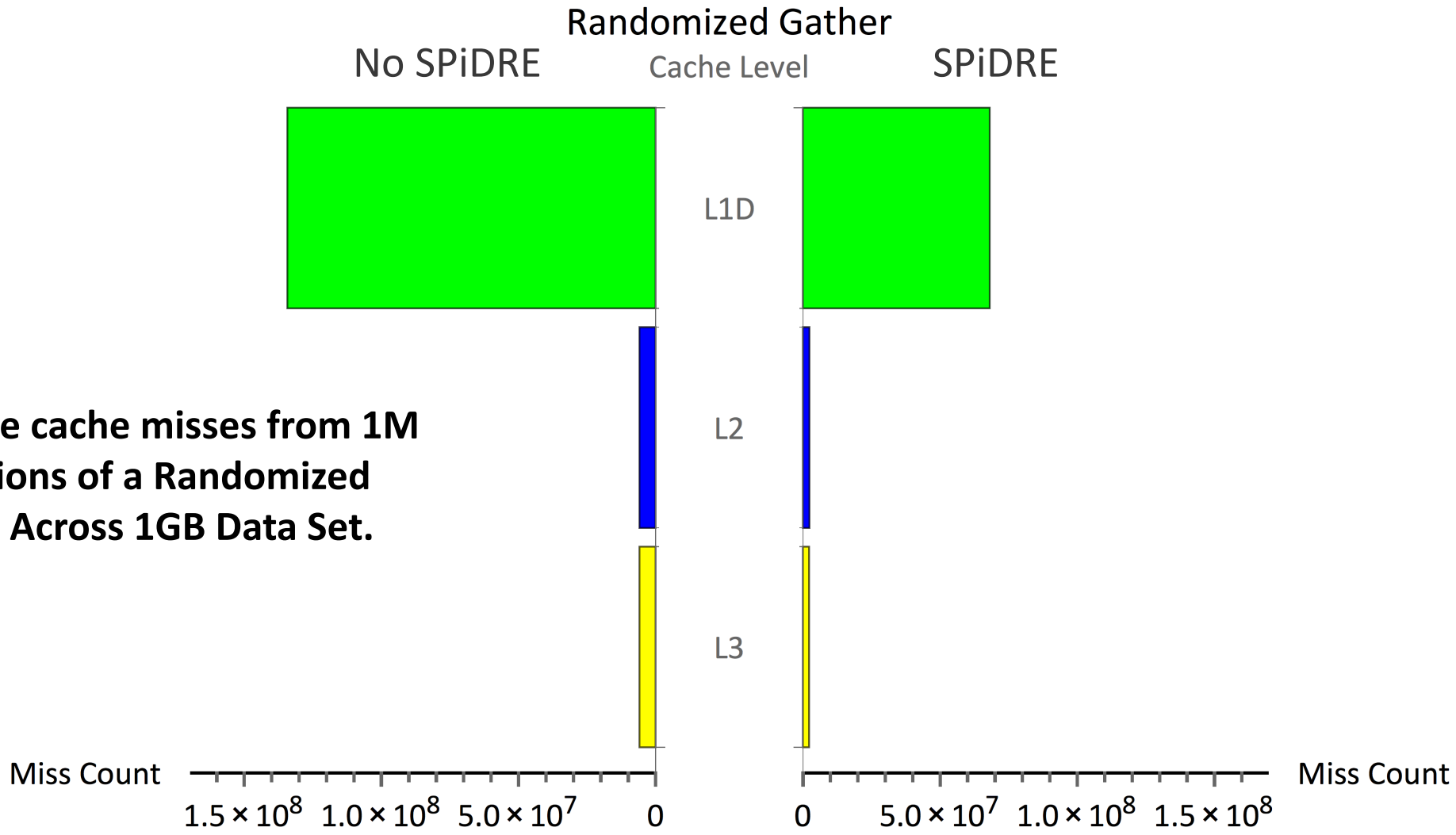
arm

Fixed Stride (1GB data set) - Gather SPiDRE



arm

Random Gather - SPiDRE



Average cache misses from 1M executions of a Randomized Gather Across 1GB Data Set.

arm

HPC Mini-apps

All single threaded executions for initial study

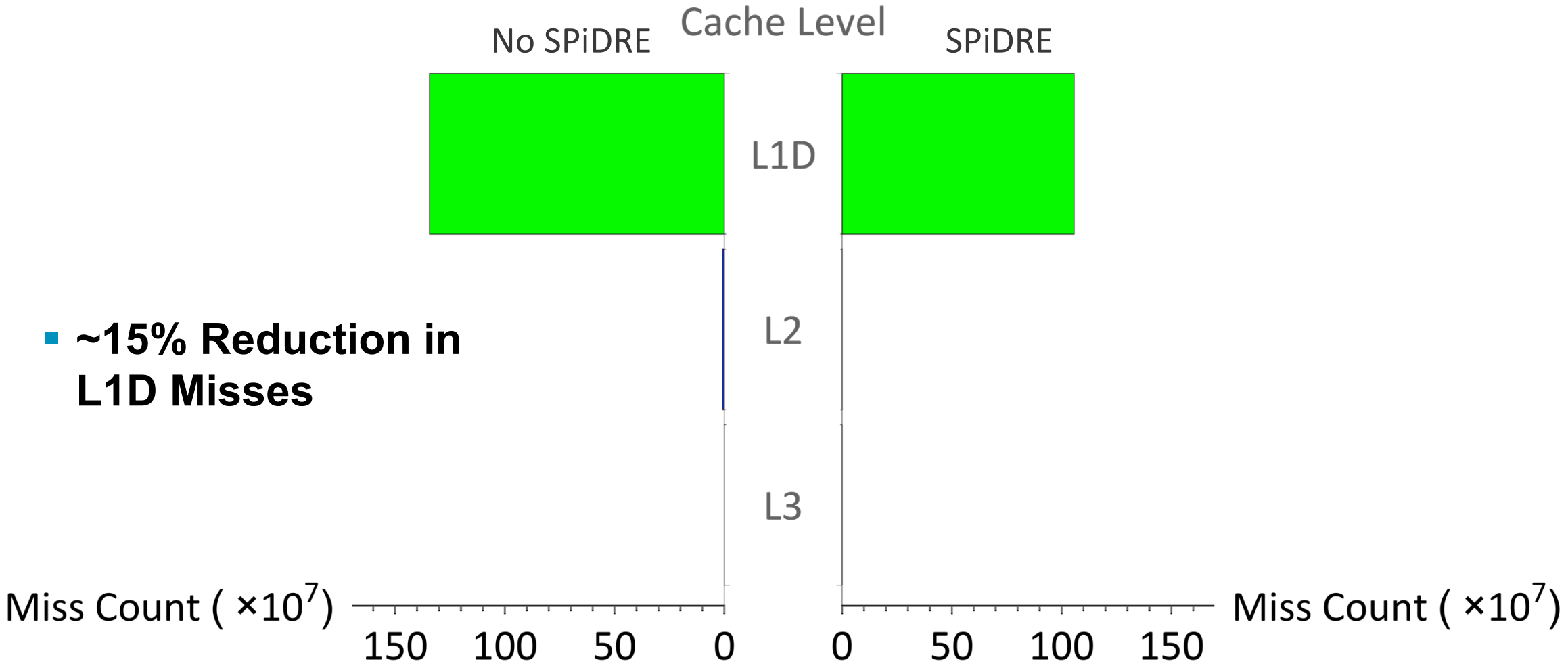
CoMD

- Used Lennard-Jones potentials
- Simple port to gather within the main loop

LULESH

- Multiple ports with varying rearrange to use distances
- Varied size and iteration count
- Demonstrated sparse data reduction combined with programmer placed pre-fetch hints

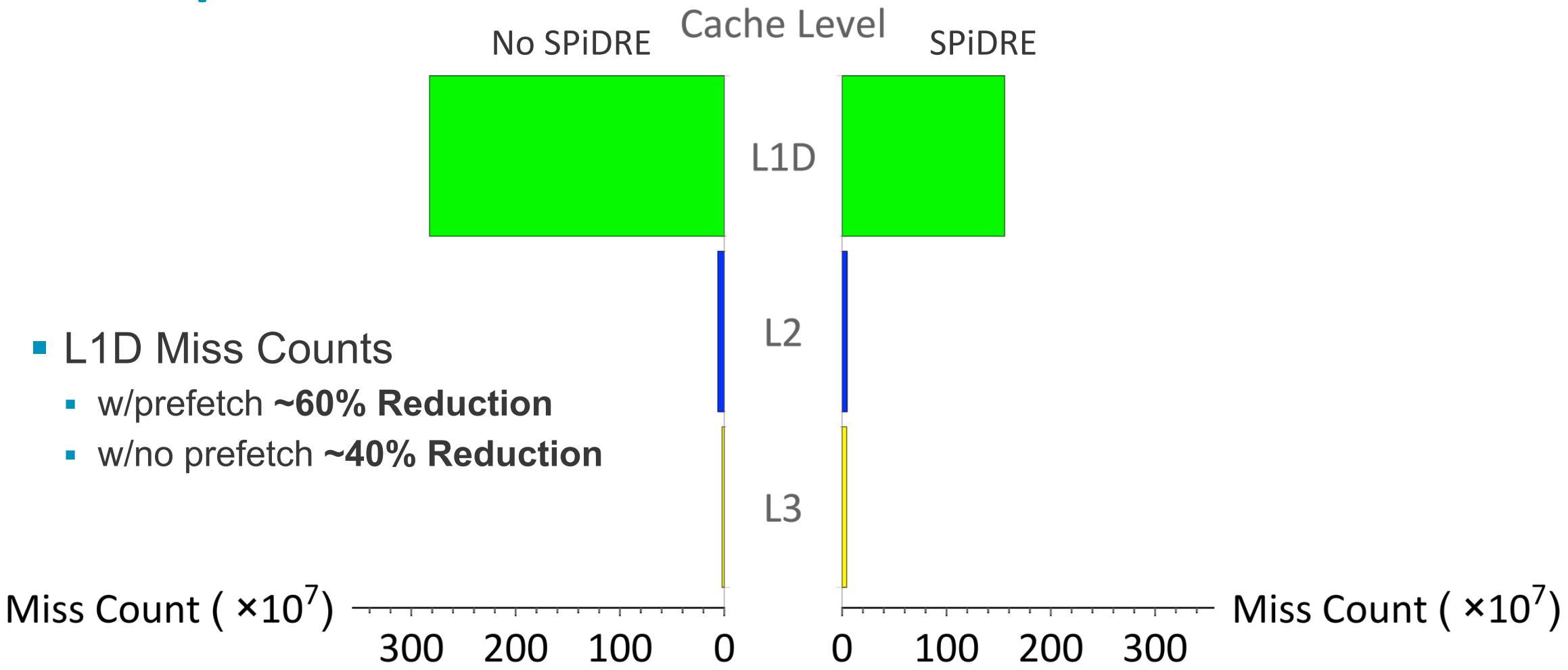
CoMD Naïve Port to SPiDRE



- **~15% Reduction in L1D Misses**

arm

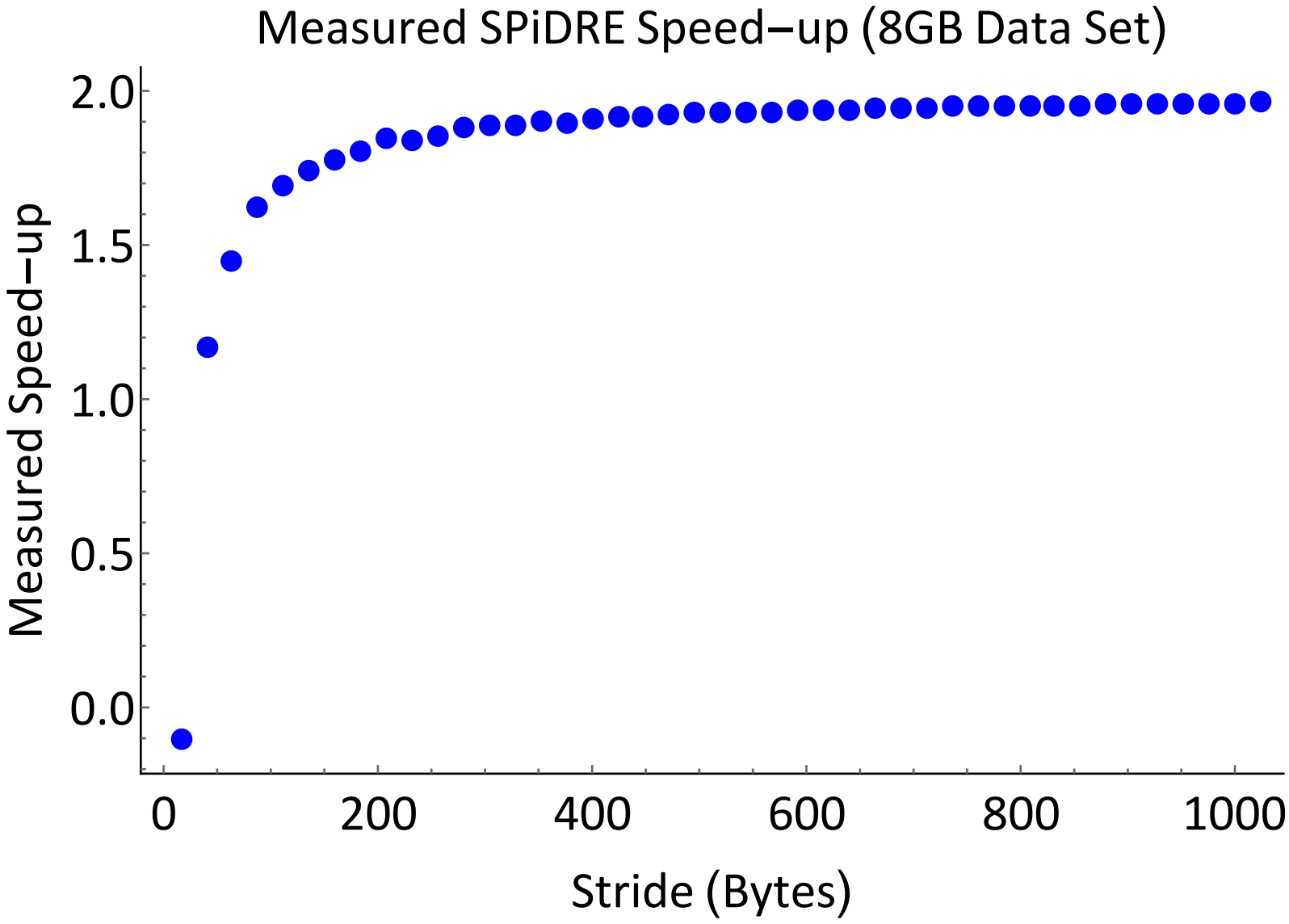
LULESH ported to SPiDRE



- L1D Miss Counts
  - w/prefetch **~60% Reduction**
  - w/no prefetch **~40% Reduction**

arm

Speed-up by average stride



arm

Conclusions

- SPiDRE is an interface and hardware acceleration infrastructure to gather data near memory/storage and make it dense (reducing bandwidth utilization, enabling more vectorization)
- We've shown a 2x speedup and significant data movement reduction on several applications, definitely more room (some special cases greater than 2x)

Future work

- Translation for near-memory compute (inc. gather/scatter)
- Programming models
- Full system simulation

Questions…

Twitter: @jonathan_beard
Email: jonathan.beard@arm.com