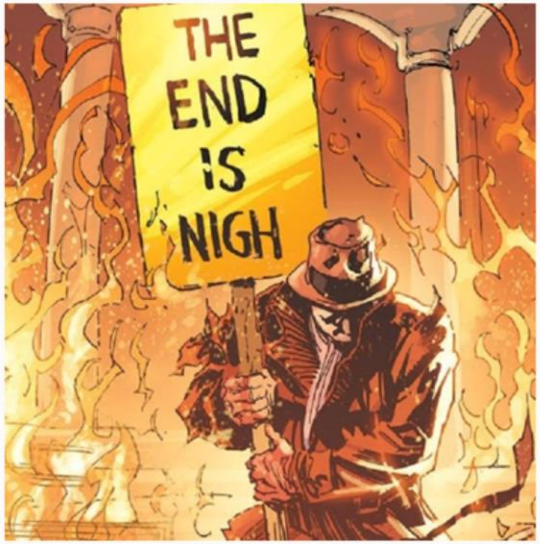
Optimizing BLIS for a NUMA Architecture:
A glimpse of the coming NUMApocalypse

Leick Robinson
Oracle

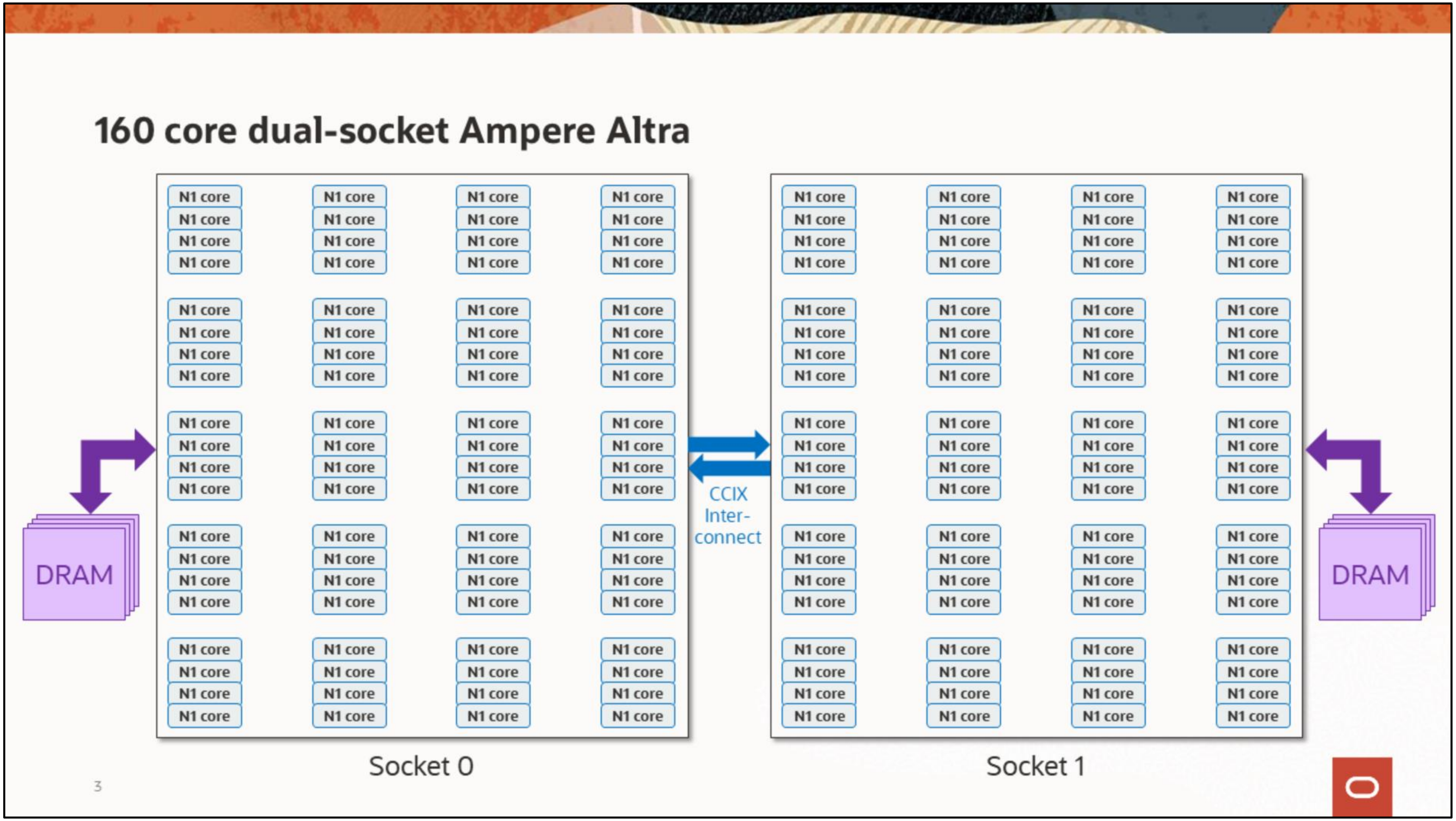
BLIS Retreat
Sept 23, 2022

What is the NUMApocalypse?

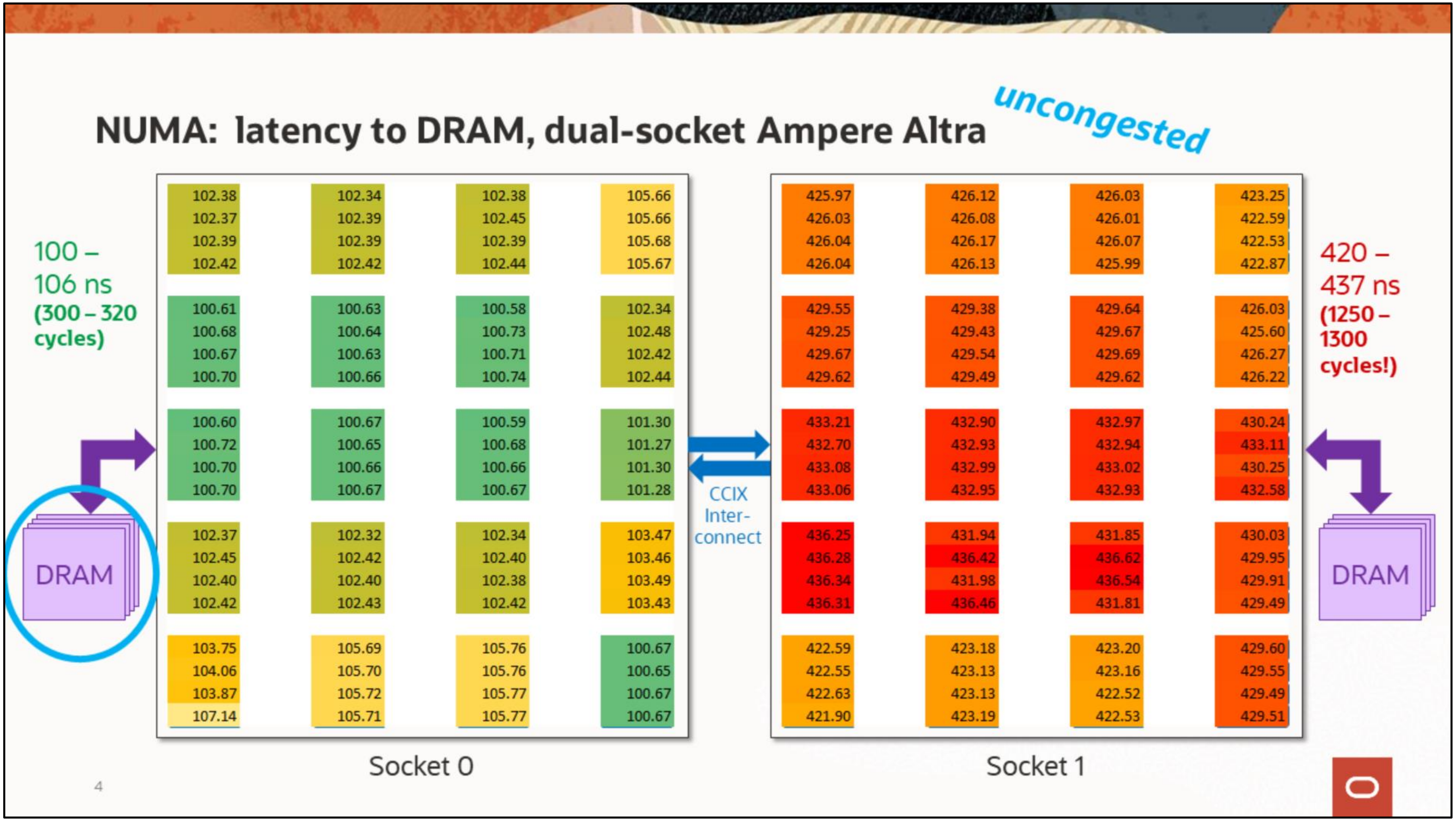


- A fun, catchy title
- More cores! More memory!
- (Moore's Law is no longer doing the heavy lifting)
- Uniform access means slow access for everyone
- You get the best performance when you intermix cores and memory
  - Scalability
  - Non-uniform memory access (NUMA)
- The trend is toward increasingly skewed NUMA architectures
  - This makes programming for performance increasingly complex
- The NUMA architecture we're exploring today is a harbinger of the future!

2



160 Neoverse N1 ARM cores on 2 sockets. 1 thread pinned to each core.

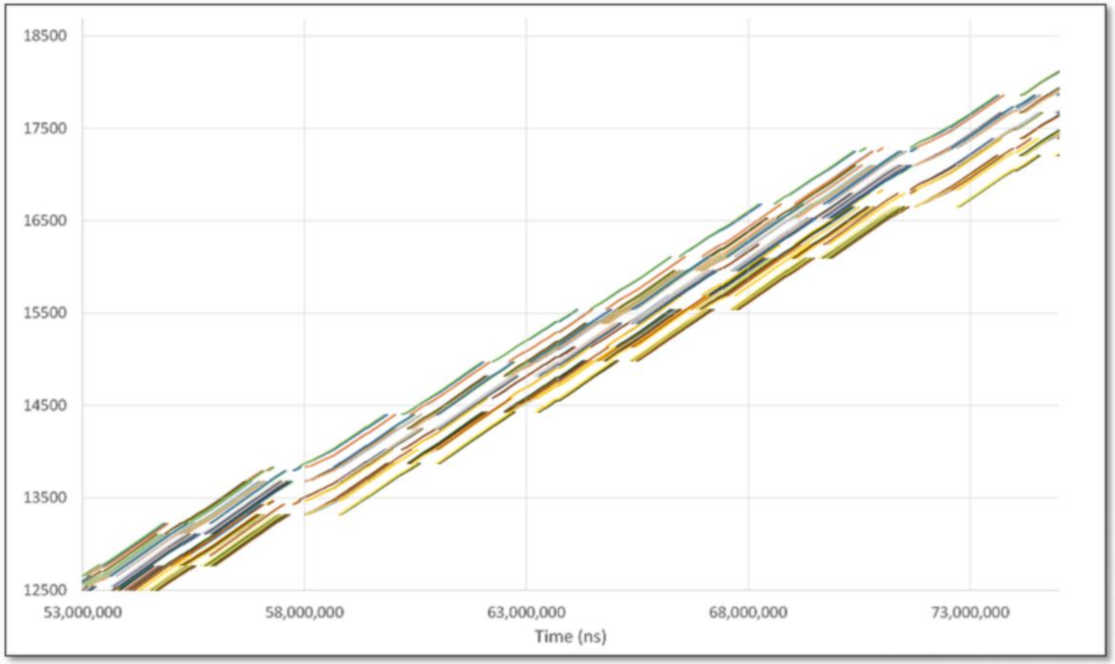


We measured the latencies from each core to the DRAM on socket 0, and we can express it as a heat map.
We can see from this that there are essentially 2 NUMA regions, one for each socket.
We can also see the variation within each zone due to location on the chip.
On socket 0, the cores near the center of the chip have lower latency because the memory controllers are near the center of the chip.
On the other hand, on socket 1, we see the reverse pattern. Cores near the center of the chip have worse latency because the CCIX interconnects are located at the corners of the chip.
Important note: this are the *uncongested* latencies. (we'll revisit that later)

sgemm, 7216 x 7216 (m = n = k = 7216)

- The problem that we'll be using as an example today is an sgemm calculation using 7216x7216 square matrices.
- In this example problem, we obtained ~5 TFLOPS, but we have been able to achieve over 6 TFLOPS.

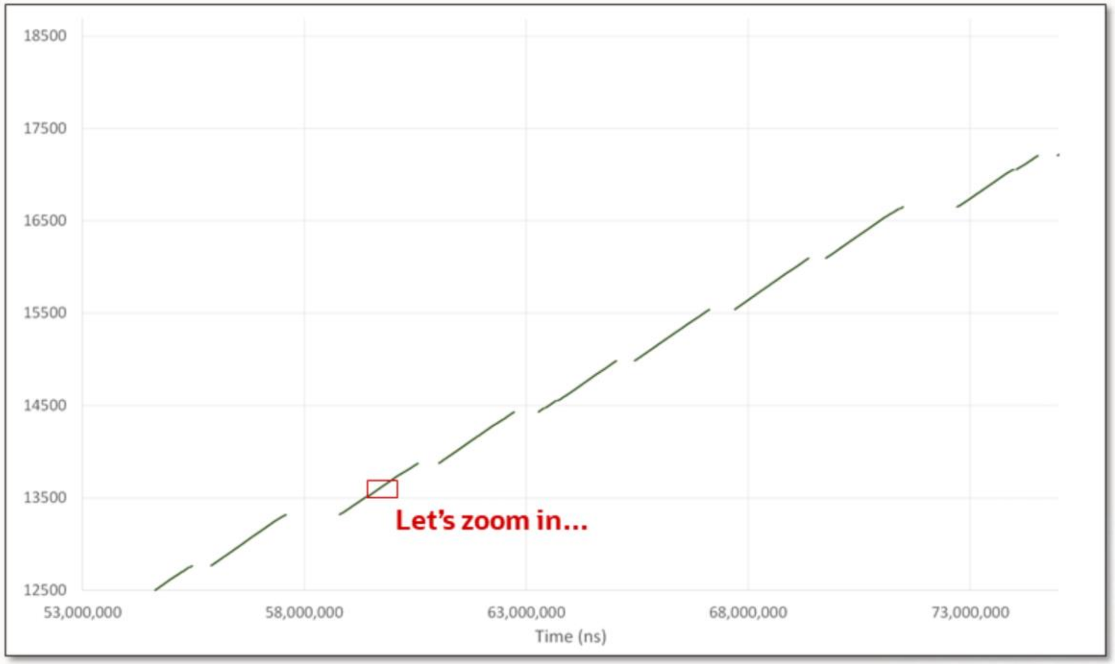
Sleet graph – sgemm, 7216 x 7216



- Allows simultaneous visualization of the activity of all 160 threads on one graph.
- Represents "work done" as a function of time.

Introduce the "sleet graph" concept.

Sleet graph – sgemm, 7216 x 7216

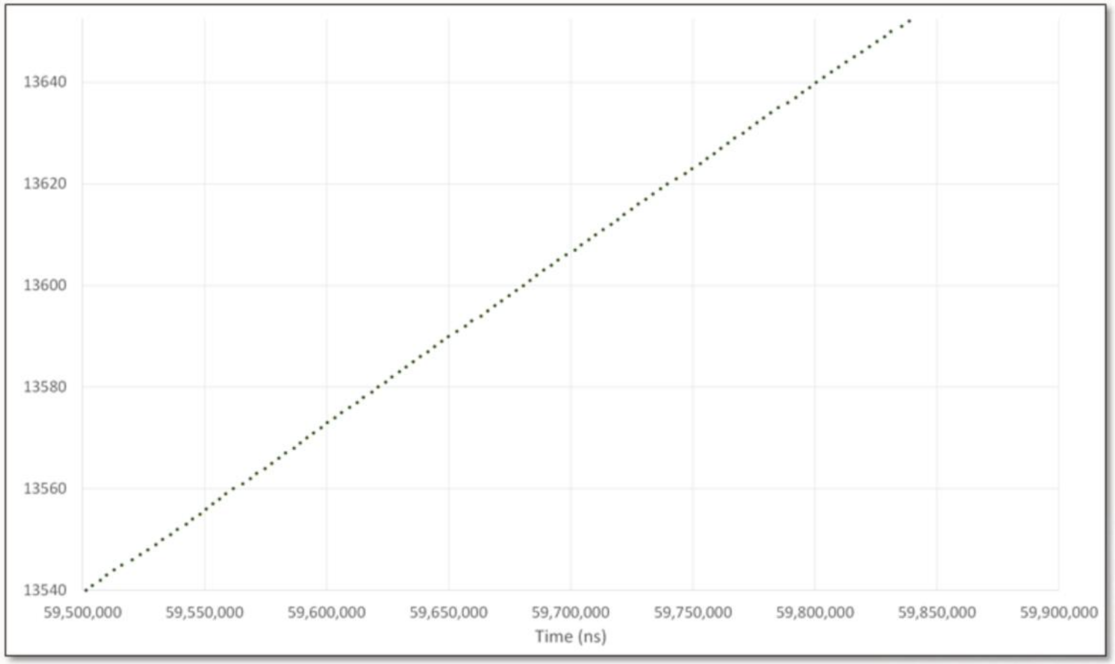


- Allows simultaneous visualization of the activity of all 160 threads on one graph.
- Represents "work done" as a function of time.

7

To better understand this, let's isolate just one of the threads, and zoom in on the indicated section.

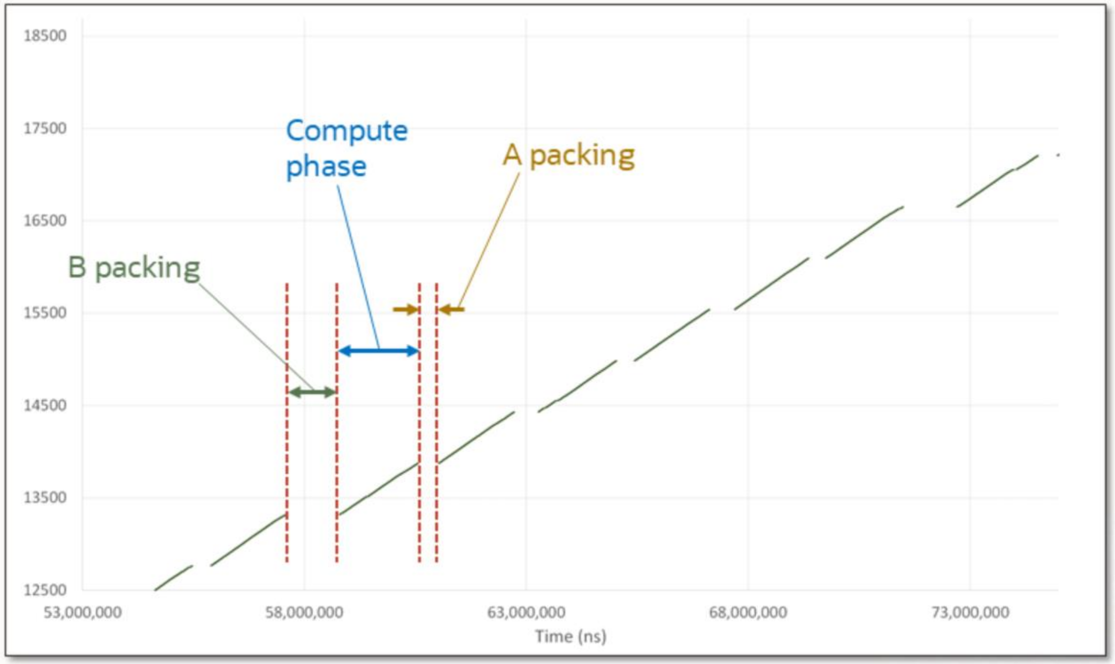
Sleet graph – sgemm, 7216 x 7216



- Allows simultaneous visualization of the activity of all 160 threads on one graph.
- Represents "work done" as a function of time.
- Each dot represents one execution of the compute microkernel. (This is our metric of "work")

So, the vertical axis is just a count of how many times this thread has invoked the compute microkernel.

Sleet graph – sgemm, 7216 x 7216

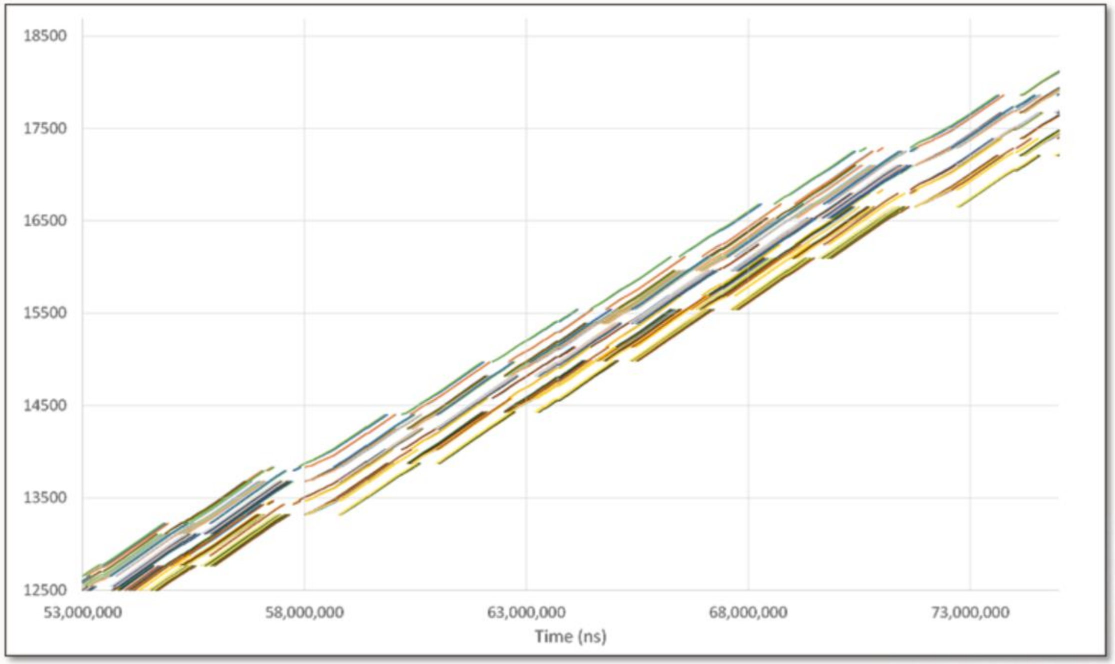


- We can clearly visualize the compute and packing phases

9

Zooming back out…

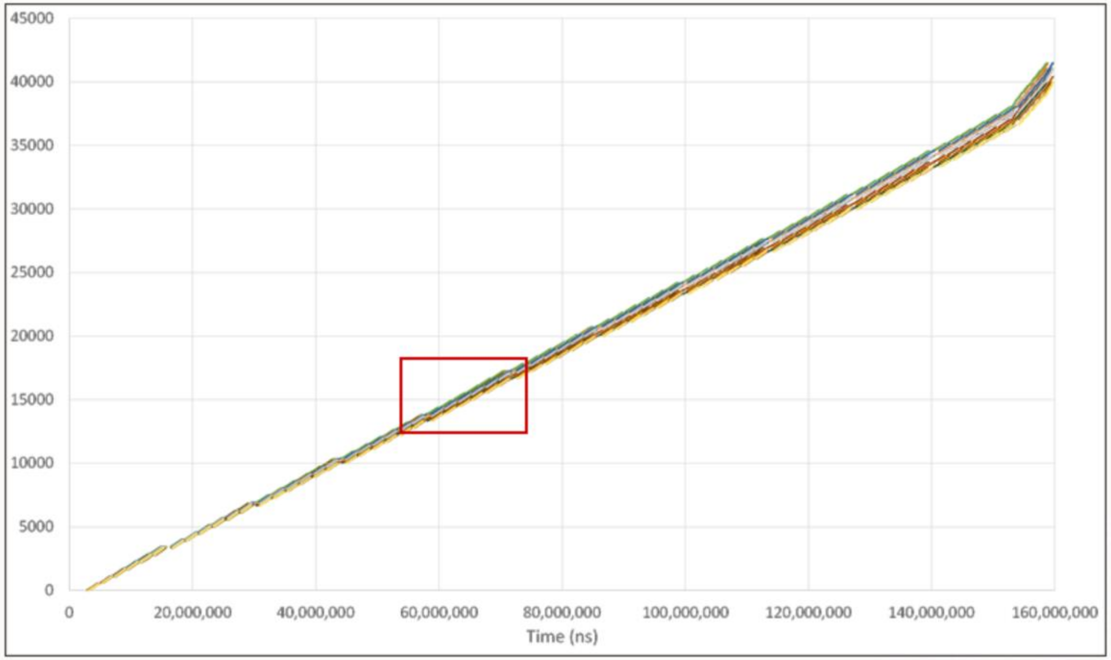
Sleet graph – sgemm, 7216 x 7216



- You may be wondering, why the dispersion?
- The work can't always be equally distributed among the threads.
  - (particularly at the "edges")
- Every compute phase, the little bit of extra work is always given to the same threads, which causes the threads to spread out over time.

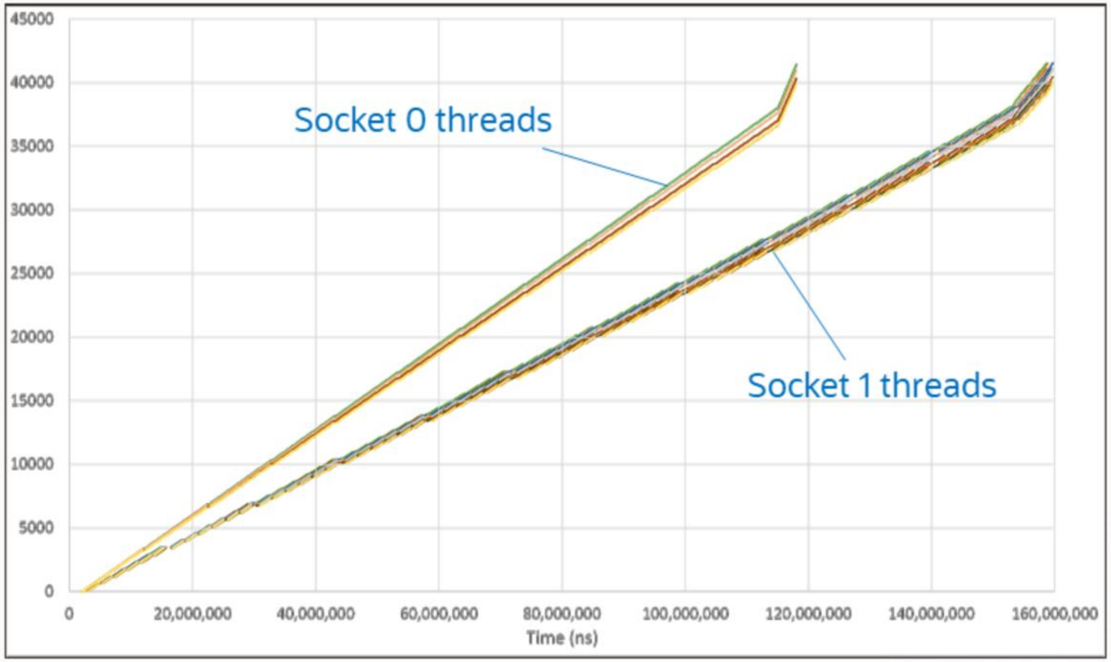
(simply because some threads are consistently doing a little bit more work than the other threads)

Sleet graph – sgemm, 7216 x 7216



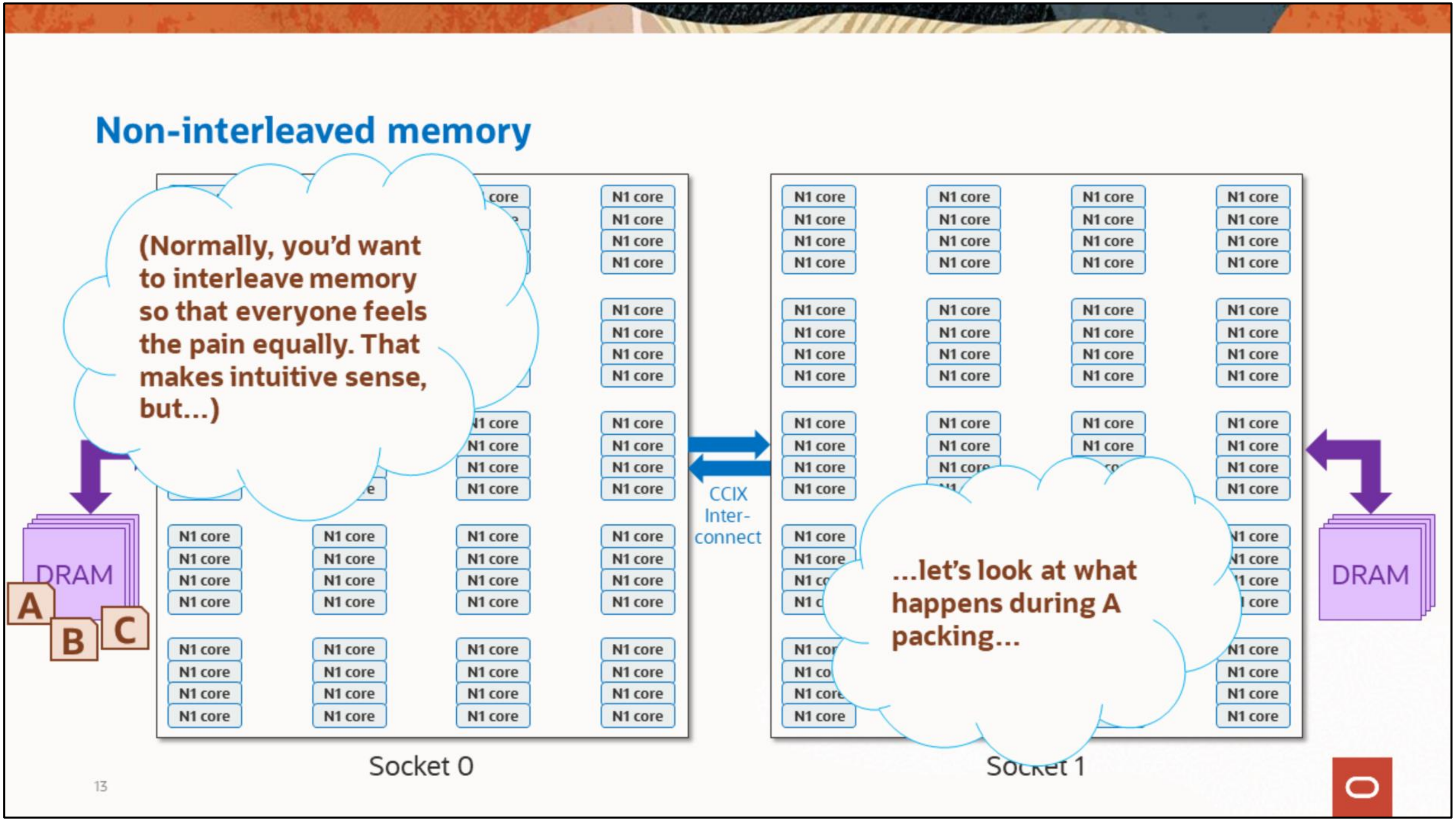
- Zooming out to the entire run:
- The calculation takes ~160 ms in total
- We can see the dispersion steadily increases over the course of the run, as you would expect.

Sleet graph – sgemm, 7216 x 7216

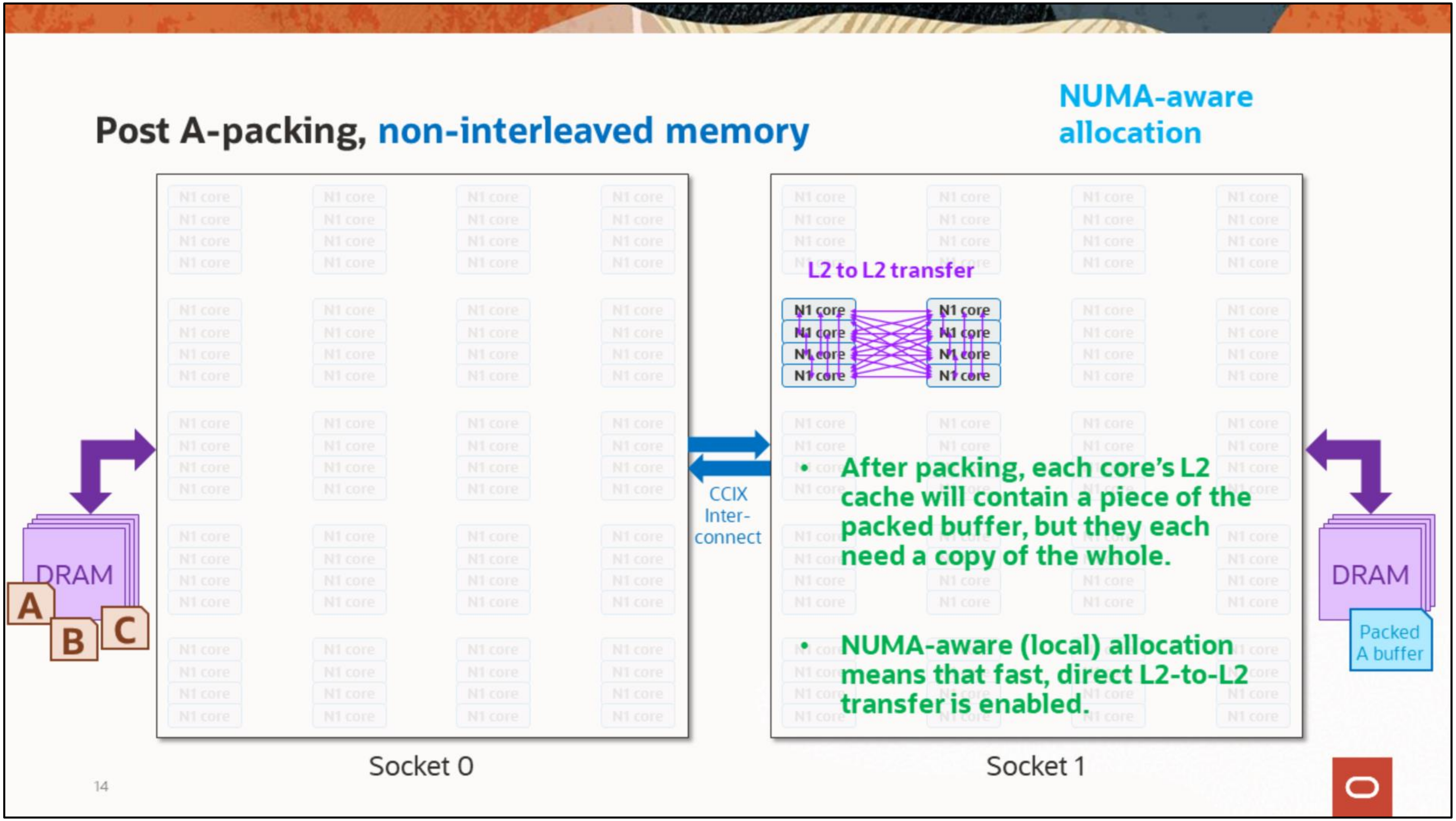


- But those were just the socket 1 threads. Here we see the threads from both sockets.
- Why the asymmetry?
- Let's talk about memory interleaving…

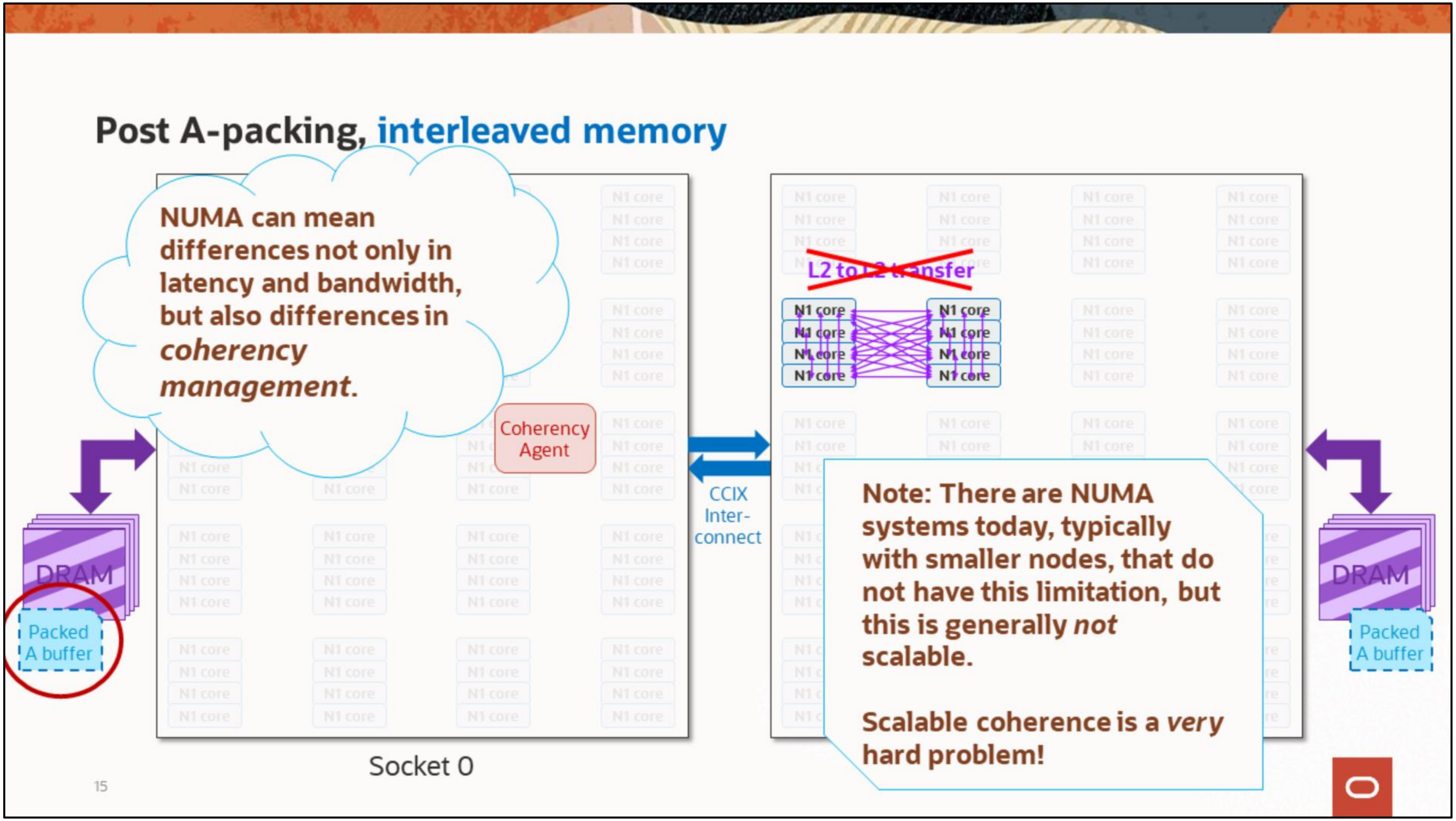
12



If memory is not interleaved, then the operand matrices (A, B, & C) would most likely all live on socket 0's DRAM, because that's where the primary thread that created them most likely lives. But, that leads to an unfair asymmetry: the cores on socket 1 are always experiencing longer latencies, while the cores on socket 0 are always experiencing shorter latencies.

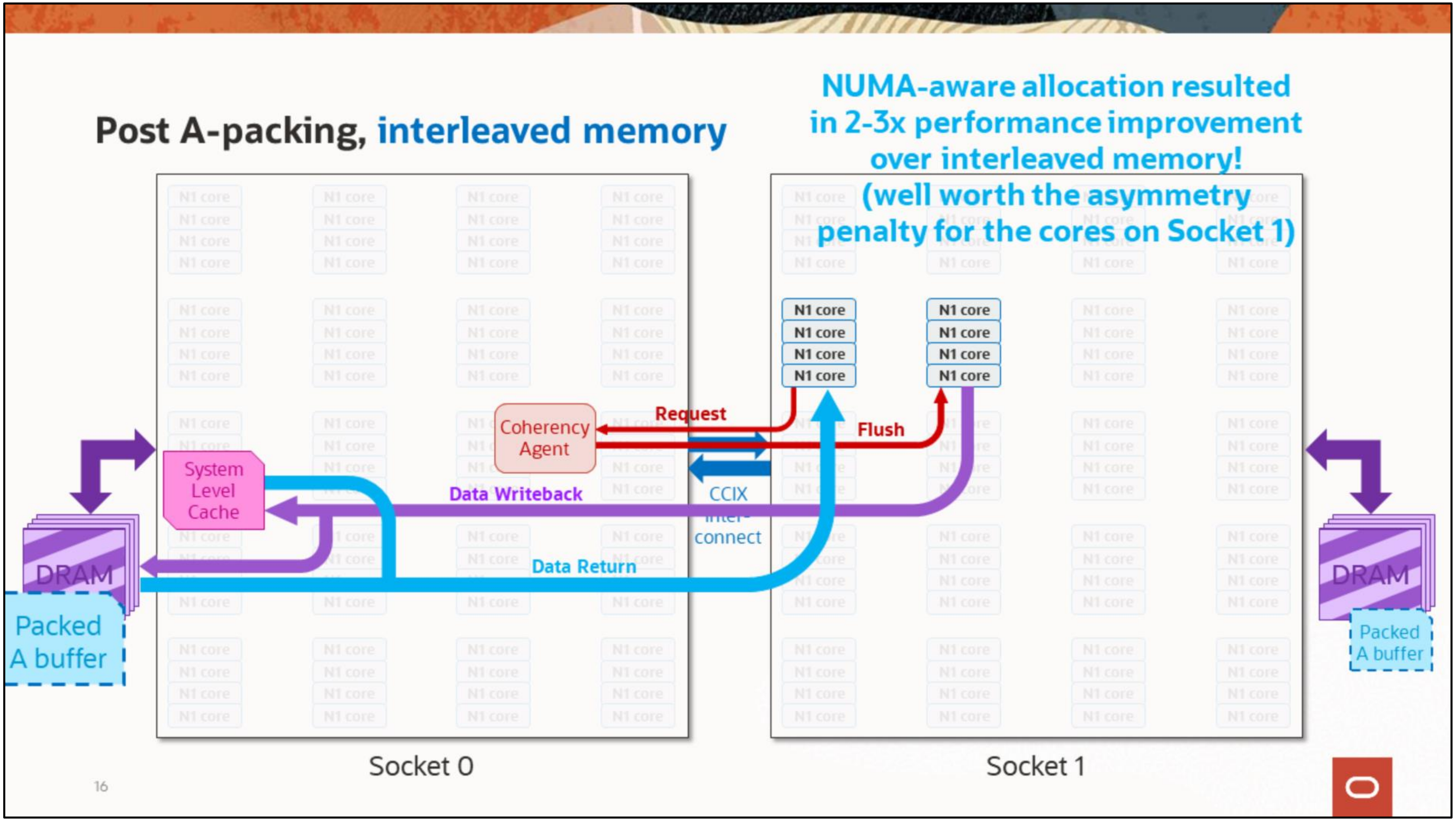


During A packing, a group of threads like that shown collaborate to pack their shared "packed A" buffer.
With NUMA-aware allocation, the buffer will be allocated to the address space of the socket 1 DRAM, since these threads all live on socket 1.



With interleaved memory, the data is striped across both DRAMS. This means that part of the packed buffer is in the socket 0 address space and part is in the socket 1 address space.

For the part that lives on socket 0, because the *global coherency point* for those addresses is on socket 0, from the point of view of socket 1 it lives *somewhere else*, and so local L2-to-L2 transfers are not allowed.



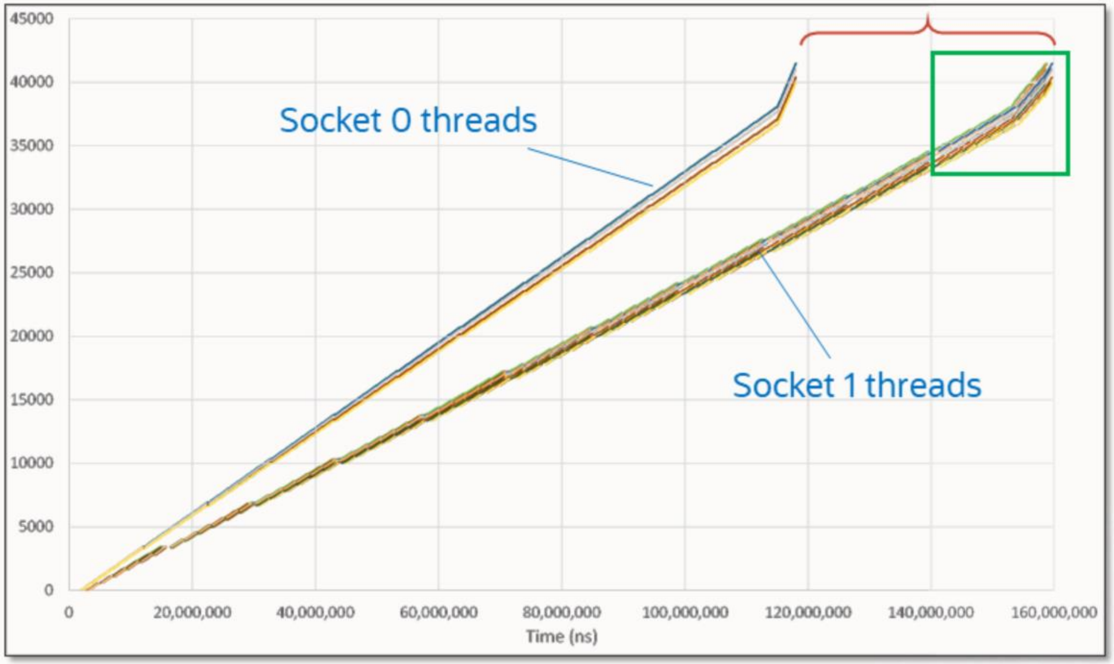
Instead, for these cores to "trade" data with their neighbors, each cache line has to get written all the way out to DRAM or SLC on the other socket, and then read back.
This is *very* slow!

First Takeaway…

1. Memory control: (canonical) interleaving can be problematic (scalability and coherence)
   - Local allocation gives better performance (requires NUMA-aware memory pools)
     - *But*, access to the source arrays is asymmetric, so one socket is “favored”.
   - *Better idea:* finer memory control: main arrays interleaved, but working arrays local!

Sleet graph – sgemm, 7216 x 7216

All the threads on socket 0 are idle here



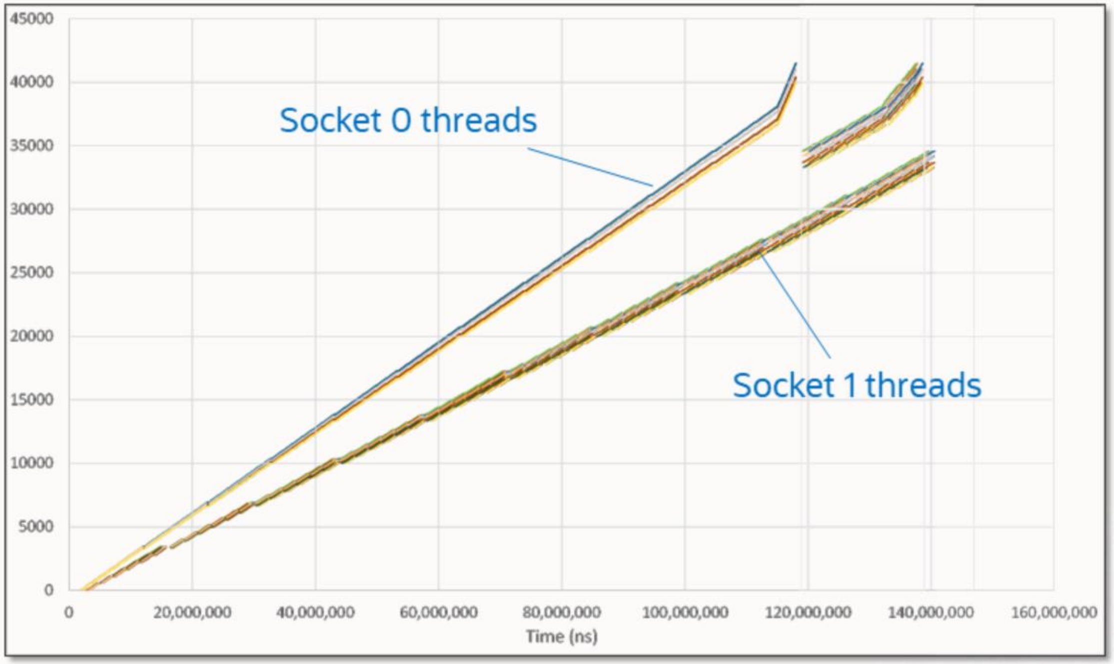
- NUMA breaks the assumption that "equal work takes equal time".
- Low-level task sharing is problematic.
  - Thread affinity
  - Cache affinity
- But, non-static distribution of work at the outermost loops (jc and pc) might be feasible.

18

But, this asymmetry means that all the threads on socket 0 finish and then sit idle.

Dynamic work distribution at the outermost loops would mean, that, for example, these last two iterations (shown by the green box) could be moved to the threads on socket 0.

Sleet graph – sgemm, 7216 x 7216



- NUMA breaks the assumption that "equal work takes equal time".
- Low-level task sharing is problematic.
  - Thread affinity
  - Cache affinity
- But, non-static distribution of work at the outermost loops (jc and pc) might be feasible.
- Here, would give an overall 14% speedup!

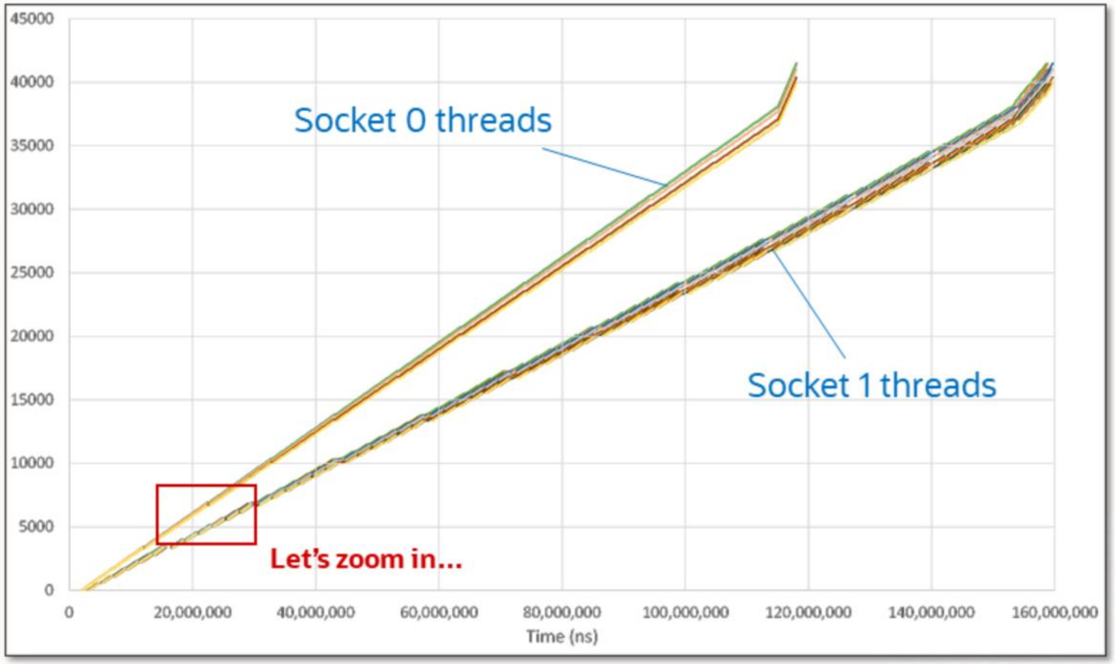
19

Takeaway #2

1. Memory control: (canonical) interleaving can be problematic (scalability and coherence)
    - Local allocation gives better performance (requires NUMA-aware memory pools)
        - *But*, access to the source arrays is asymmetric, so one socket is "favored".
    - *Better idea:* finer memory control: main arrays interleaved, but working arrays local!

2. NUMA-"friendly" load balancing
    - One idea: Coarse-grained task stealing (at the outermost loops) might be feasible

Sleet graph – sgemm, 7216 x 7216

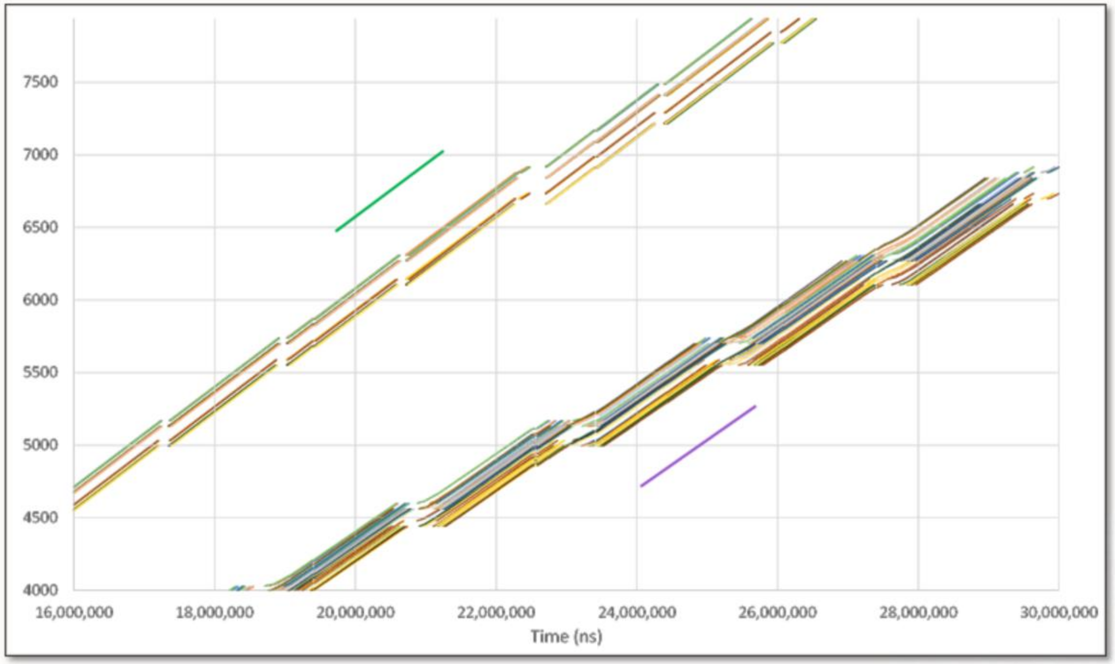


- But where is this performance difference coming from?

21

Let's zoom in to find out…

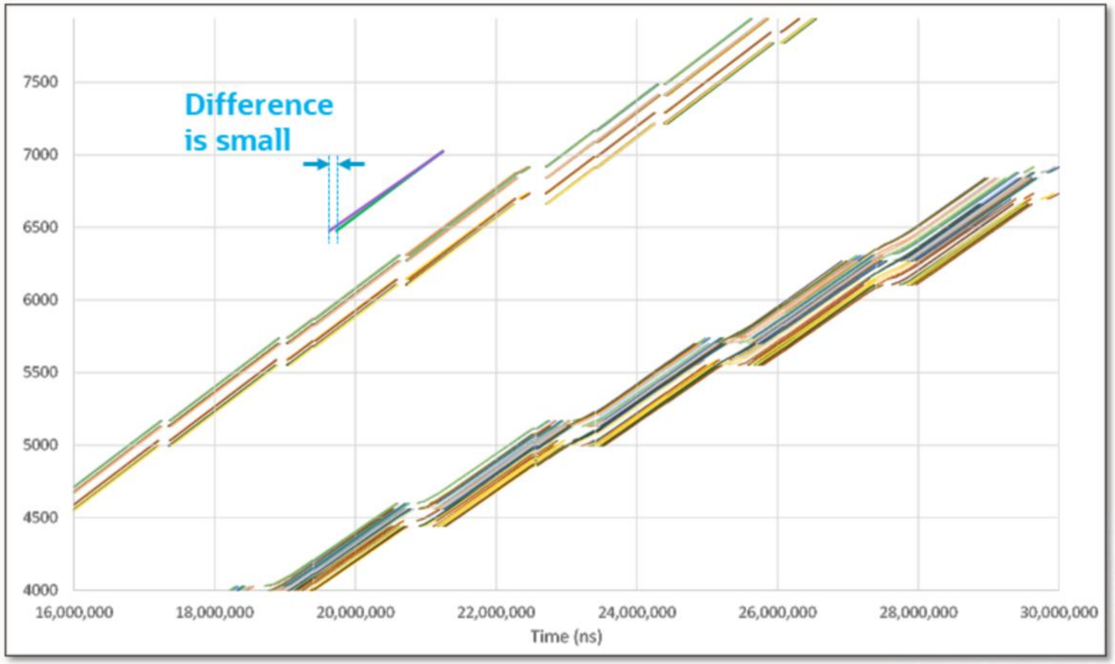
Sleet graph – sgemm, 7216 x 7216



- But where is this performance difference coming from?
- The slopes of the compute phases give the relative compute performance.

I.e., the slopes of the “bars”.

Sleet graph – sgemm, 7216 x 7216

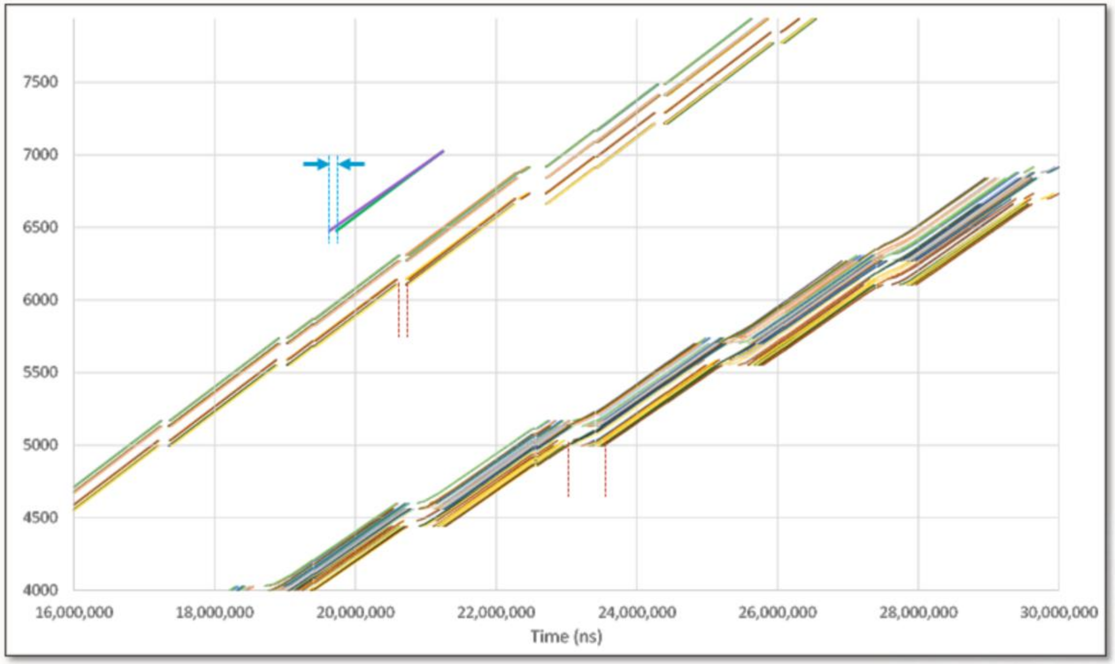


- But where is this performance difference coming from?
- The slopes of the compute phases give the relative compute performance.
- Compute phase is near maximum efficiency
  - ~9000 cycles, *should be* plenty of time to prefetch C microtile

23

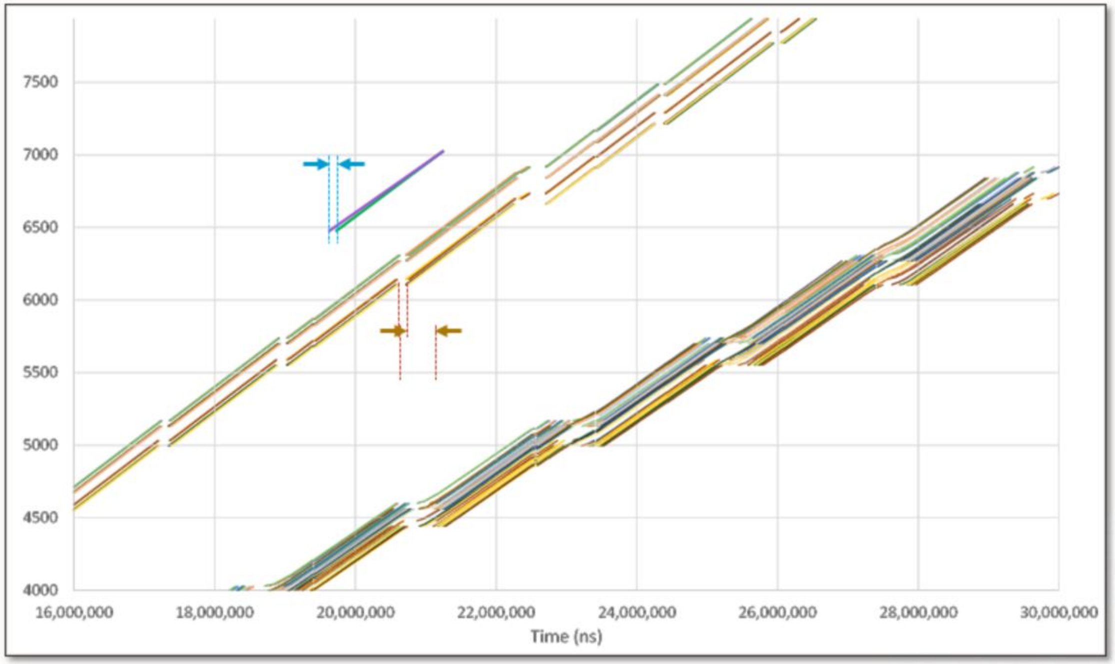
But put a pin in that "*should be*". We'll revisit that later.

Sleet graph – sgemm, 7216 x 7216



- But where is this performance difference coming from?
- The slopes of the compute phases give the relative compute performance.
- Compute phase is near maximum efficiency
  - ~9000 cycles, *should be* plenty of time to prefetch C microtile
- More significant is the difference in A packing time.

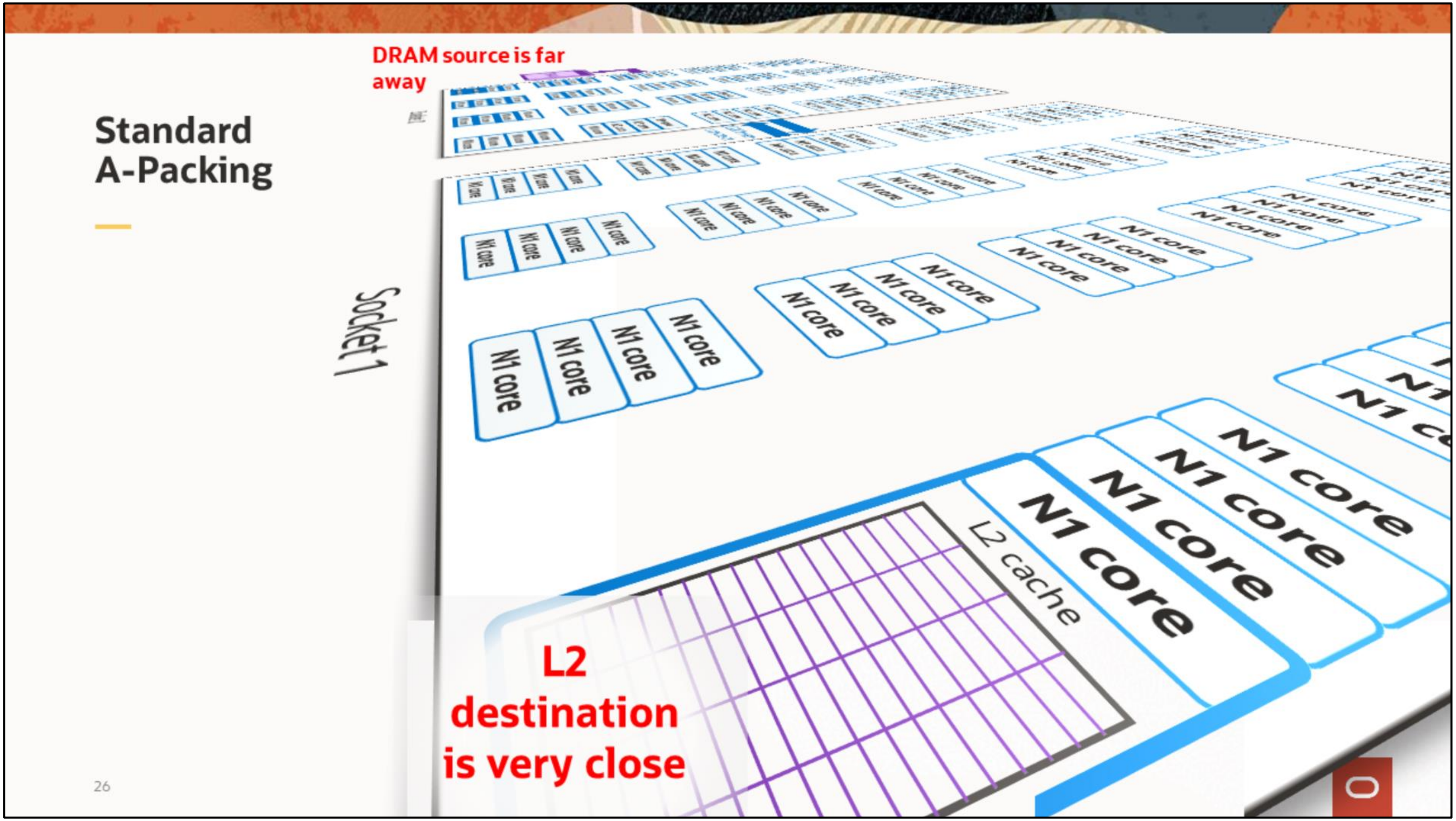
Sleet graph – sgemm, 7216 x 7216



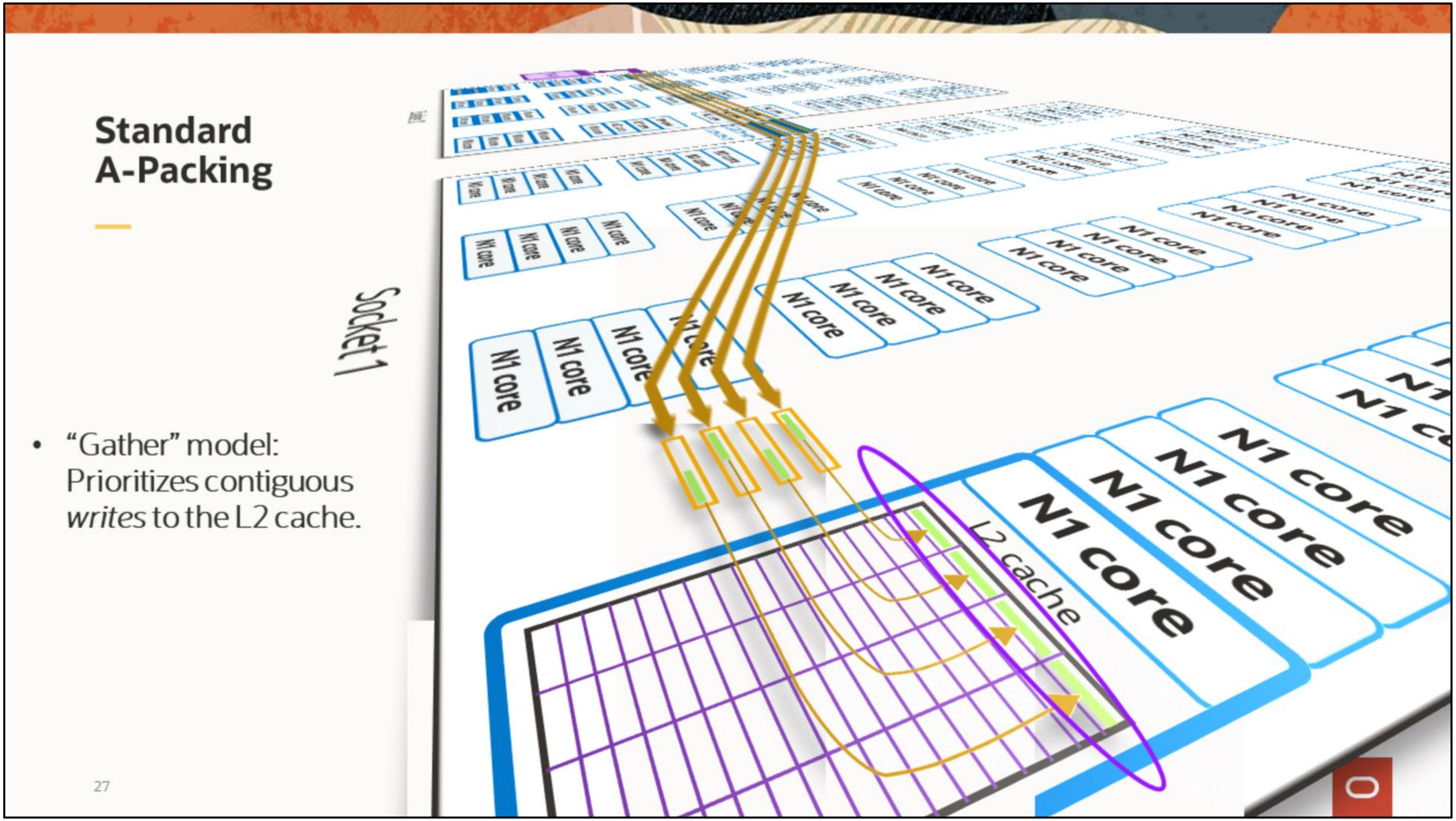
- But where is this performance difference coming from?
- The slopes of the compute phases give the relative compute performance.
- Compute phase is near maximum efficiency
  - ~9000 cycles, *should be* plenty of time to prefetch C microtile
- More significant is the difference in A packing time.

25

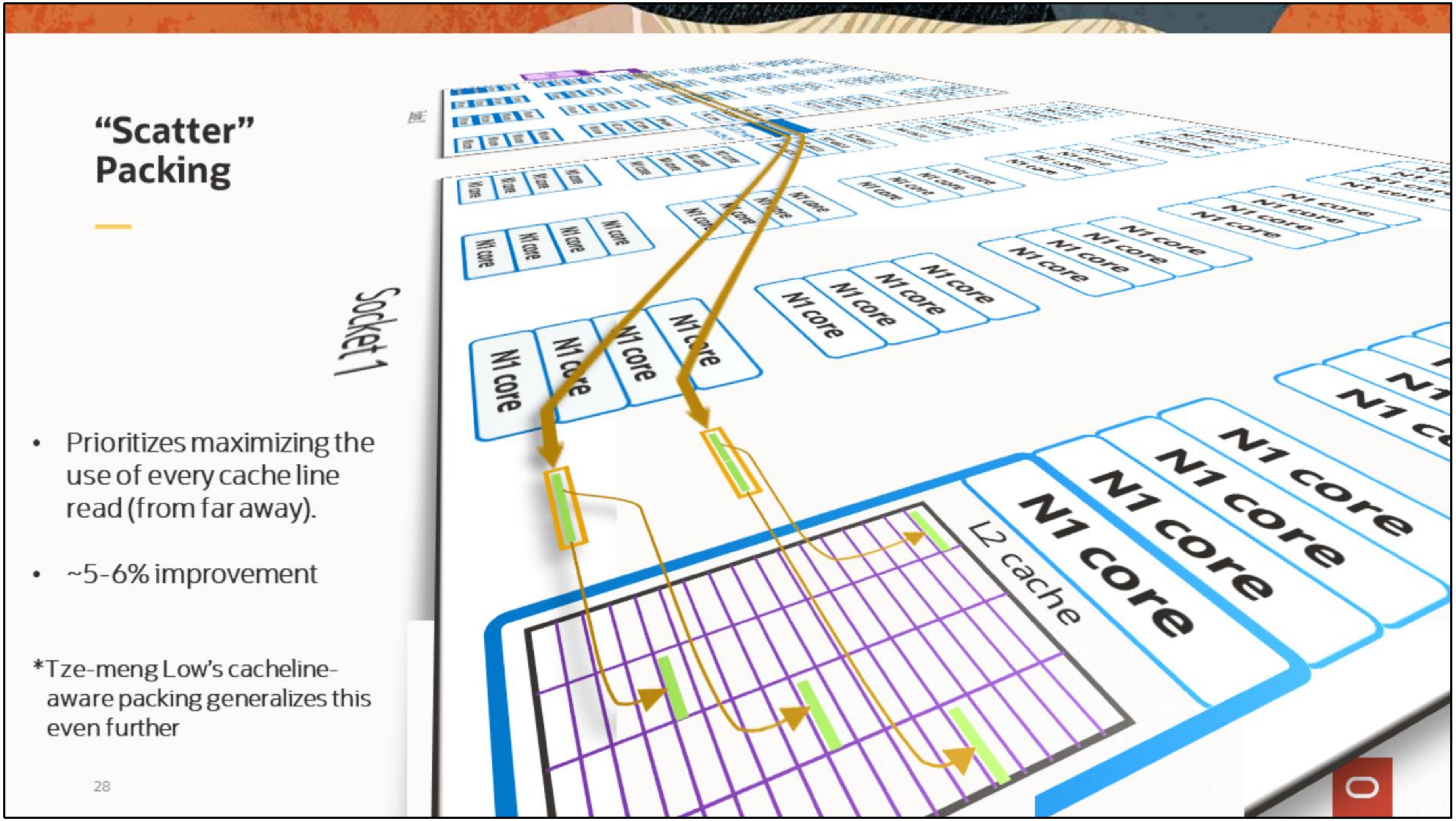
So, let's talk about what's involved with A packing…



During A packing, the cores on socket 1 are reading data from the socket 0 DRAM which is *far away*, and writing to their own L2 cache which is *very close by*.



Standard packing in BLIS today follows a “gather” model, that prioritizes contiguous *writes*.



We've instead experimented with a "scatter" packing model, that prioritizes contiguous *reads*.

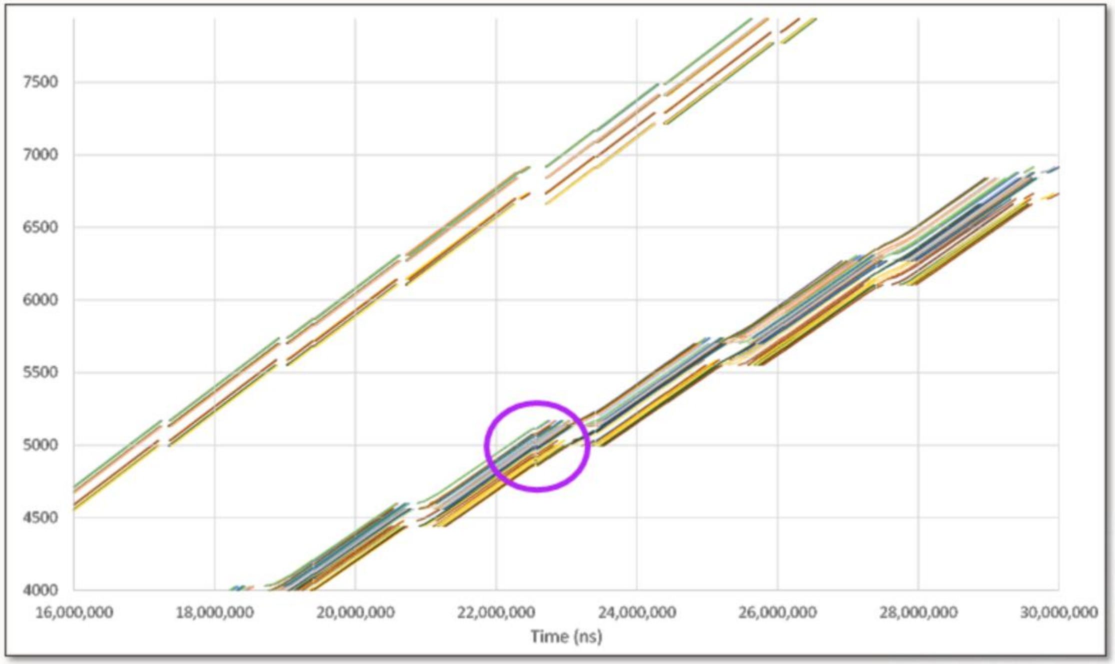
*Shout out to Tze-Meng. Since doing this work, I became aware that he has been independently pursuing the same idea. However, he has generalized the concept even further to what he calls "cacheline-aware" packing.

Takeaway #3

1. Memory control: (canonical) interleaving can be problematic (scalability and coherence)
   - Local allocation gives better performance (requires NUMA-aware memory pools)
     - *But*, access to the source arrays is asymmetric, so one socket is "favored".
   - *Better idea:* finer memory control: main arrays interleaved, but working arrays local!

2. NUMA-"friendly" load balancing
   - One idea: Coarse-grained task stealing (at the outermost loops) might be feasible
3. NUMA-cognizant utilization (the data you're reading may come from far away)
   - NUMA-aware packing (e.g., scatter packing or cacheline-aware packing)
   - Future: May also need NUMA-friendly packing kernels with *much* deeper prefetching

Sleet graph – sgemm, 7216 x 7216

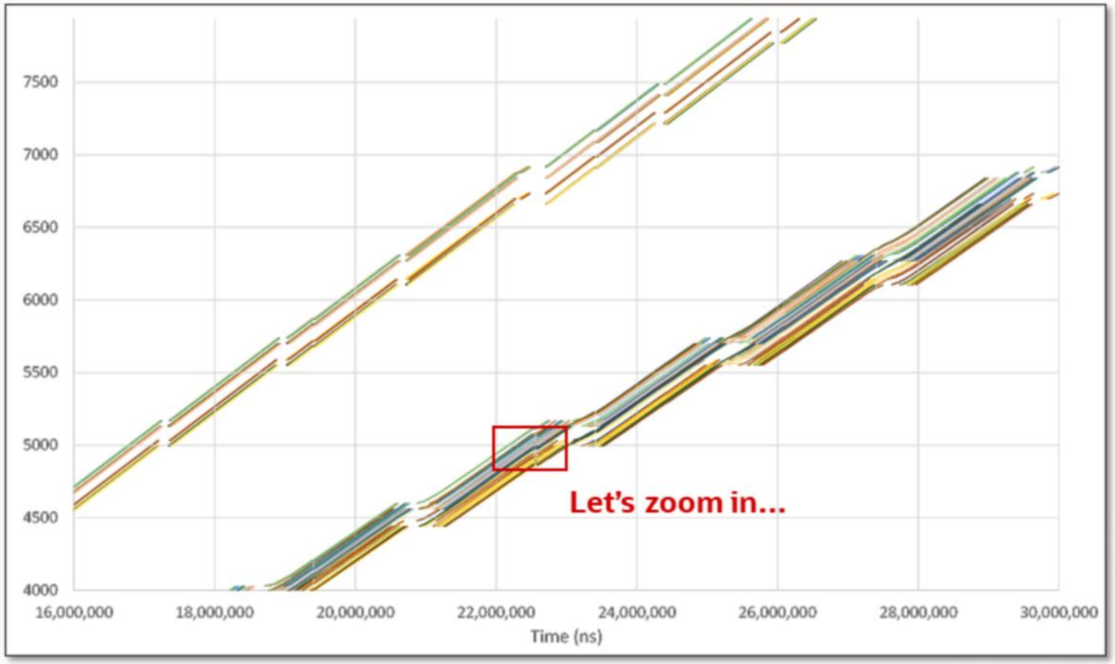


- What's happening here?

30

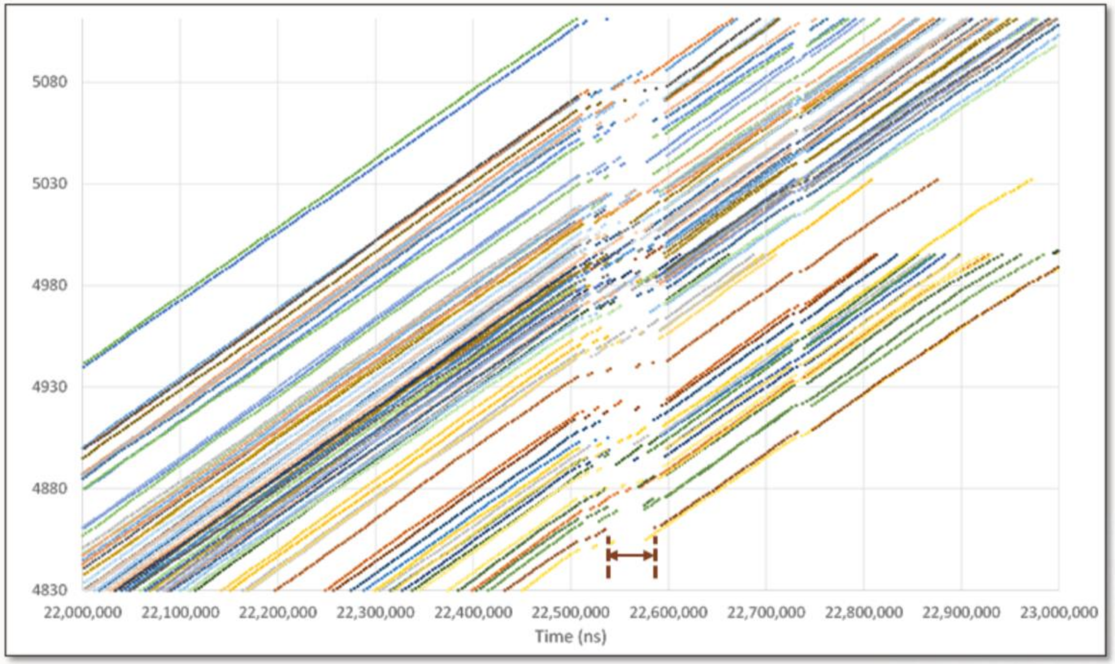
The eagle-eyed among you may have noticed this and wondered…

Sleet graph – sgemm, 7216 x 7216



- What's happening here?

Sleet graph – sgemm, 7216 x 7216

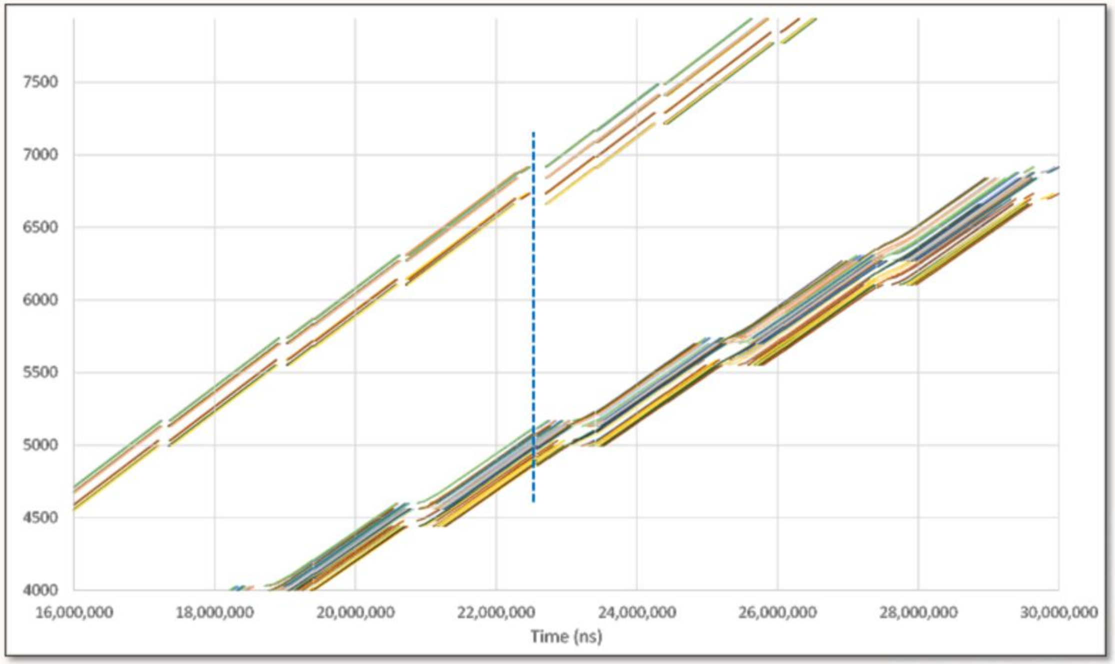


- All of the threads on socket 1 are stalling waiting for their prefetched C microtile data.
- Up to 150,000 cycles! Why?

32

Note: this event spans about a tenth of a millisecond, so ~300,000 cycles.

Sleet graph – sgemm, 7216 x 7216



- Coincides with start of the B packing phase on socket 0
- Socket 0 congestion. (All of the threads are streaming data from DRAM and streaming data back out to the SLC)
- So *much* congestion that socket 1 threads can't get their data!
- Example of interference between NUMA regions.
- One option: decouple the sockets via *data replication* (every thread sees a "nearby" copy of A, B, & C)
  - BLIS L3 operations are ~N^3, while replication is only ~N^2

33

If we zoom back out, we see that this coincides with the start of the B packing phase on socket 0.
During this phase, all of the threads on socket 0 are streaming data from DRAM and back out to the SLC, producing massive congestion on the socket 0 mesh.
Since the work required for these BLIS L3 operations are of order N3, and data replication is of order N2, this means that this will *always* be cost effective above a certain problem size (we just have to determine where that threshold lies).

Takeaway 1c

1. Memory control: (canonical) interleaving can be problematic (scalability and coherence)
   - Local allocation gives better performance (requires NUMA-aware memory pools)
     - *But*, access to the source arrays is asymmetric, so one socket is "favored".
   - *Better idea:* finer memory control: main arrays interleaved, but working arrays local!
   - Another option: data replication, so every thread sees a "nearby" copy of A, B, & C
2. NUMA-"friendly" load balancing
   - One idea: Coarse-grained task stealing (at the outermost loops) might be feasible
3. NUMA-cognizant utilization (the data you're reading may come from far away)
   - NUMA-aware packing (e.g., scatter packing or cacheline-aware packing)
   - Future: May also need NUMA-friendly packing kernels with *much* deeper prefetching

34

I've added this under memory control, because decoupling the sockets through data replication would also address the latency asymmetry.

Questions?