Benchmarking Customized Hardware/Software for Graph Analytics

Aninda Manocha
PhD Student at Princeton University
PI: Margaret Martonosi

SRC JUMP

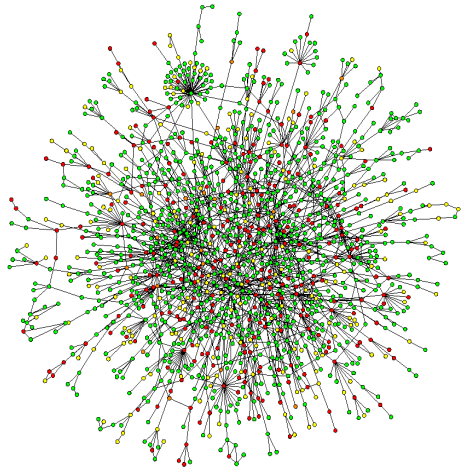
adacenter.org @ADA_Center

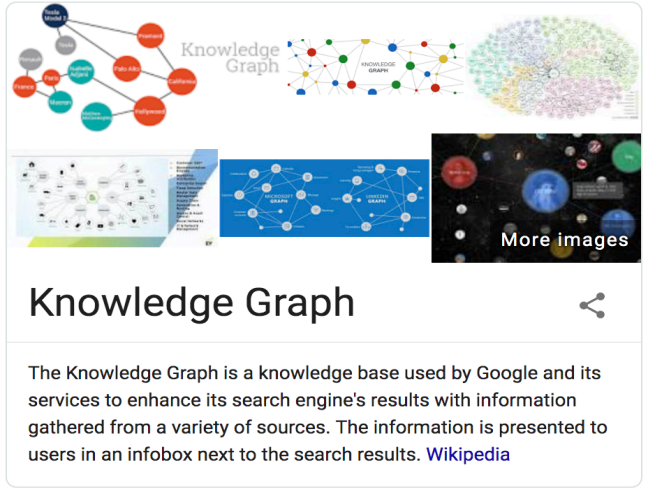
This work is supported by the Semiconductor Research Corporation (SRC) and DARPA

ada Applications Driving Architectures

Graph Applications are Emerging Workloads

- A lot of big data is naturally in the form of graphs:
  - Social networks
  - Communication networks
  - Biological networks
  - Google's Knowledge Graph
- Graph/sparse computation kernels are widely used:
  - Recommendation systems
  - Natural language processing
  - Community/cluster detection
  - Search engines (e.g. Google, Facebook)





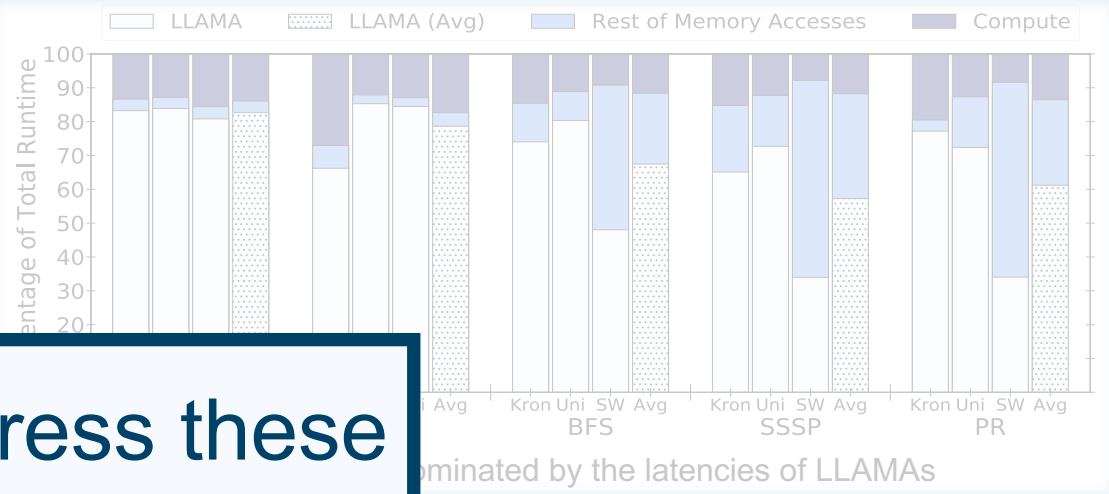
Images from MIT Sloan Management Review, https://www.smartcitiesworld.net, the European Bioinformatics Institute, and Google

PI: Martonosi

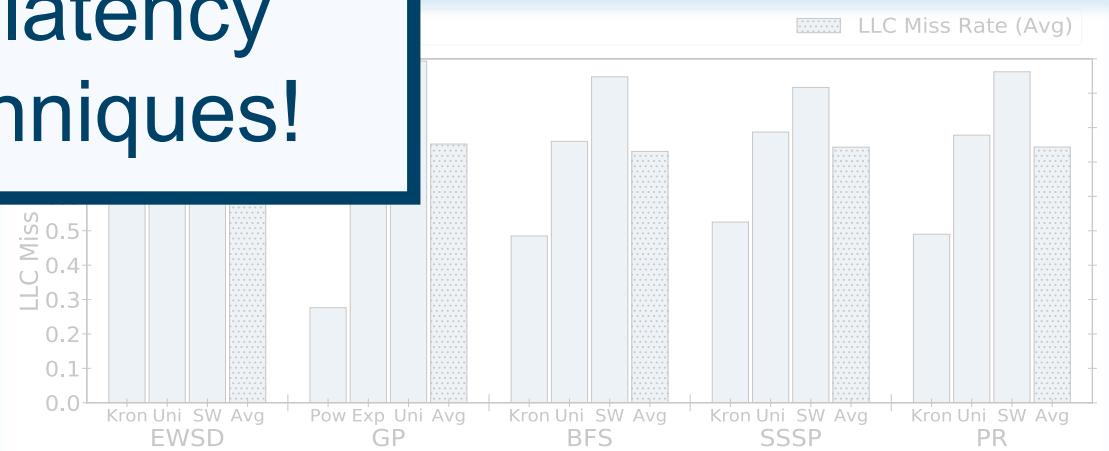
Irregular Accesses: LLAMAs in the Wild

- **Long-LAtency Memory Accesses (LLAMAs)**: irregular memory accesses in a tight inner loop
- Graph/sparse kernels have a frequent indirect access to neig
- Addressing LLAMAs a performance optimiza

Objective: Address these LLAMAs via latency tolerance techniques!



ominated by the latencies of LLAMAs



LLAMAs have severe LLC miss rates, over 50% on average

```
while (worklist_size>0)
  for (node in worklist)
    for (e in neighbors[node])
      e_idx = edge_array[e]
      v = ret[e_idx]
      if (v == not_visited)
        ret[e_idx] = hop
        worklist[wl_idx] = e_idx
        wl_idx++
  hop++
```

LLAMA

BFS Kernel Code

How can we address these LLAMAs?

- Out-of-order cores for latency tolerance mechanisms
  - Dynamic scheduling of instructions
    - Multiple instructions in flight
  - *Require roughly 10x more area and power than in-order cores* [1, 2]
- Traditional parallelism with simpler, in-order cores to speed up computation
  - *Graph/sparse applications are memory bound and have little computation*
- Decoupling computation from memory accesses?

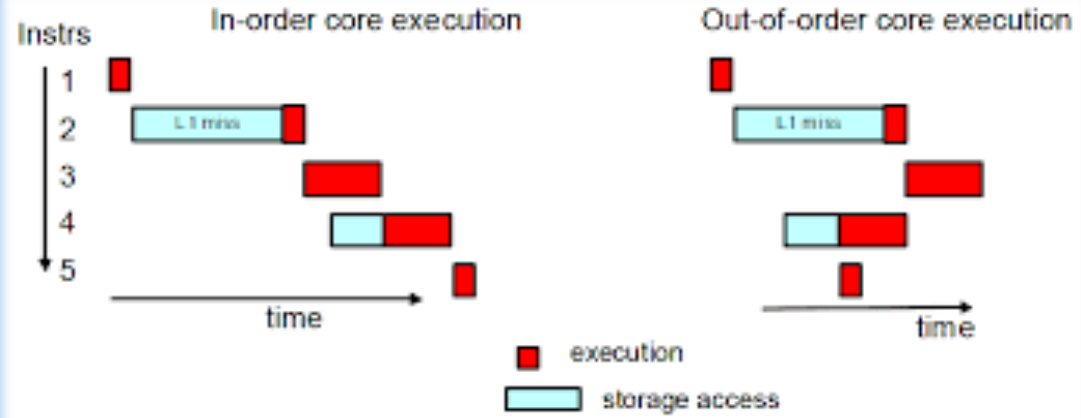


Image from http://idcp.marist.edu

Image from https://www.dkfindout.com/us/animals-and-nature/camels/llama/

[1] Asanovic et al. The Rocket Chip Generator. Techical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.
[2] Chris Celio, David A. Patterson, and Krste Asanovic, The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

What is decoupling?

- Static division of a program into two independent instruction streams:
  - *Access* [1] or *Supply* core [2]
    - Handles memory access and address computation; ”produces” data
  - *Execute* [1] or *Compute* core [2]
    - Performs computation; “consumes” data
- Ideally, the *Supply* runs ahead of the *Compute*
  - The *Supply* issues memory requests early and enqueues the data
  - Meanwhile, the *Compute* handles complex value computation, using the *Supply*’s data waiting on a queue

Original Source Code

```
while (worklist.size()>0)
  for (node in worklist)
    for (e in neighbors[node])
      e_idx = edge_array[e]
      v = ret[e_idx]
      if (v == not_visited)
        ret[e_idx] = hop
        worklist[wl_idx] = e_idx
        wl_idx++
  hop++
```

Supply

```
while (worklist.size()>0)
  for (node in worklist)
    for (e in neighbors[node])
      e_idx = LOAD(&edge_array[e])
      PRODUCE(e_idx)
      v = LOAD(&ret[e_idx])
      PRODUCE(v)
      if (v == not_visited)
        STORE_ADDR(&ret[e_idx])
        STORE_ADDR(&worklist[wl_idx])
```

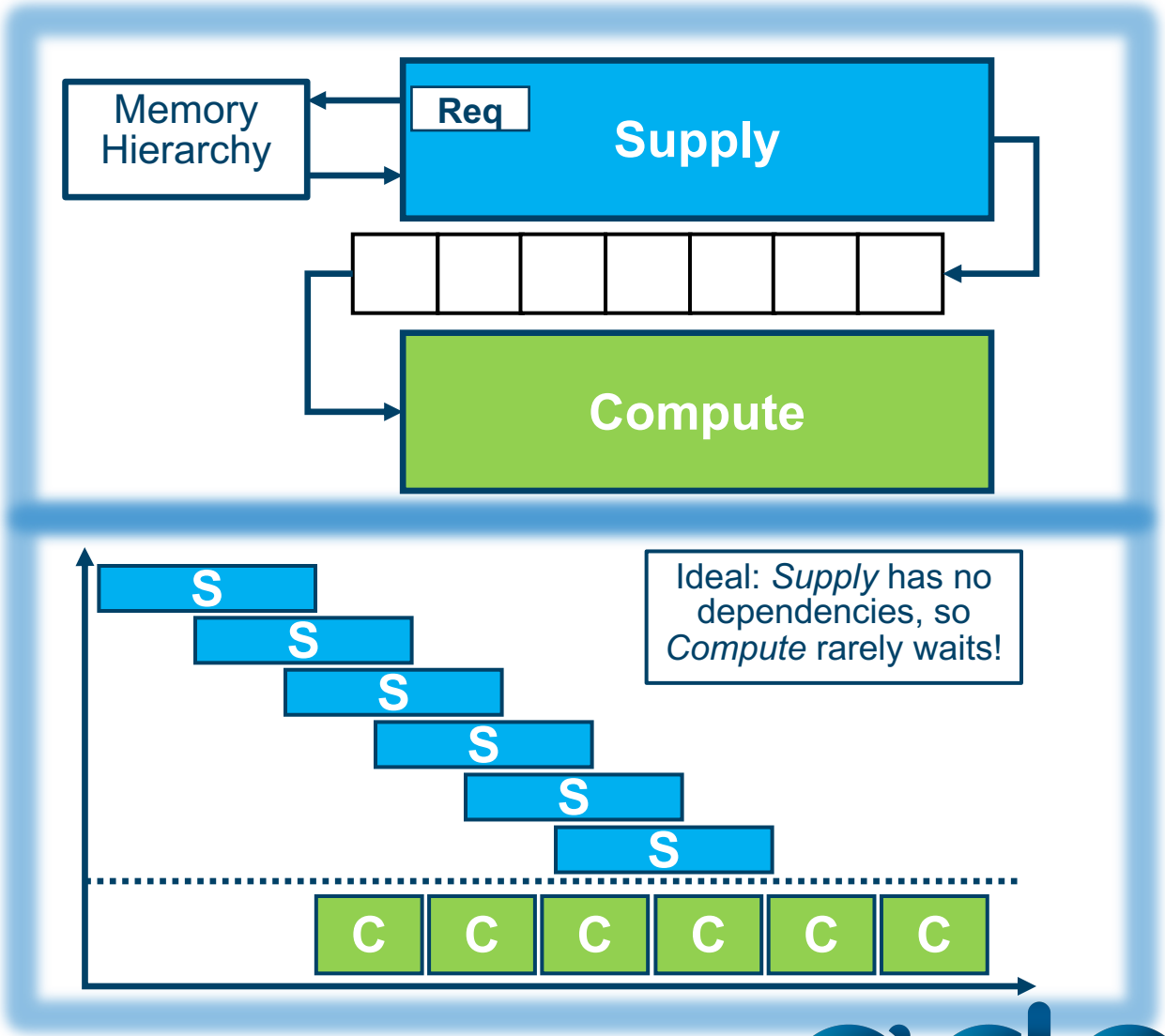
Compute

```
while (worklist.size()>0)
  for (node in worklist)
    for (e in neighbors[node])
      e_idx = CONSUME()
      v = CONSUME()
      if (v == not_visited)
        STORE_VAL(hop)
        STORE_VAL(e_idx)
        wl_idx++
  hop++
```


[1] James E. Smith. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, volume 10, pg. 112–119. IEEE Computer Society Press, 1982.
[2] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *MICRO*, pg. 191–203. ACM, 2015.

What is decoupling?

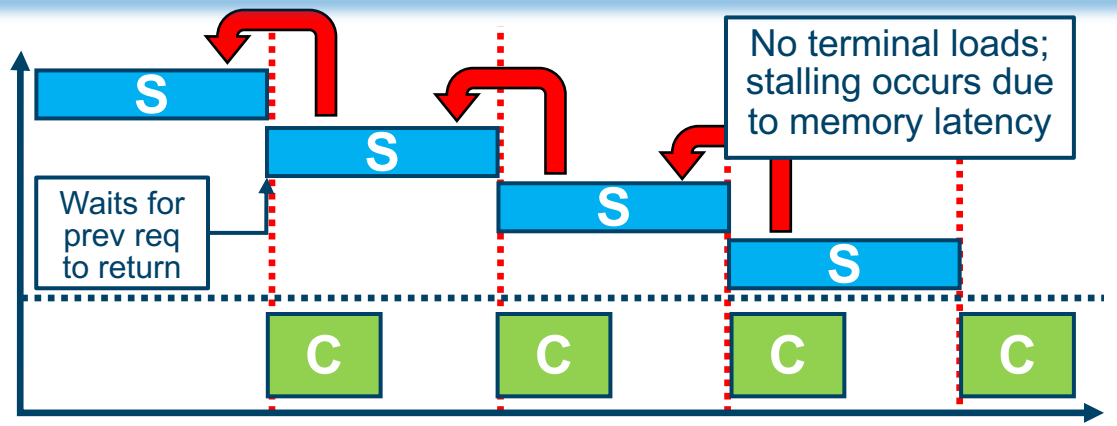
- Static division of a program into two independent instruction streams:
  - *Access* [1] or *Supply* core [2]
    - Handles memory access and address computation; ”produces” data
  - *Execute* [1] or *Compute* core [2]
    - Performs computation; “consumes” data
- Ideally, the *Supply* runs ahead of the *Compute*
  - The *Supply* issues memory requests early and enqueues the data
  - Meanwhile, the *Compute* handles complex value computation, using the *Supply*’s data waiting on a queue



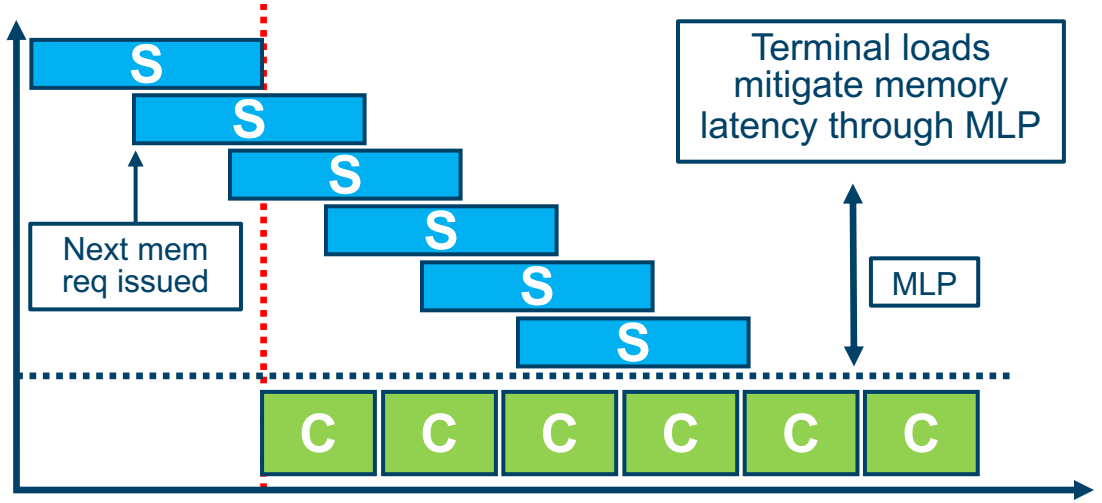
[1] James E. Smith. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, volume 10, pg. 112–119. IEEE Computer Society Press, 1982.
[2] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *MICRO*, pg. 191–203. ACM, 2015.

How does decoupling tolerate latency?

- **Terminal loads** [1]: loads that have no dependencies on the *Supply*
  - *Supply* does not have to occupy pipeline resources while waiting for memory
  - These loads can complete early
- **Terminal load buffer**: enqueues terminal loads
  - Distinction from normal loads
  - Numerous terminal loads help maintain longer *Supply* runaheads by enabling memory-level parallelism (MLP)
- Exploiting MLP mitigates memory latency effects of LLAMAs



The *Supply* issues several **non-terminal** loads

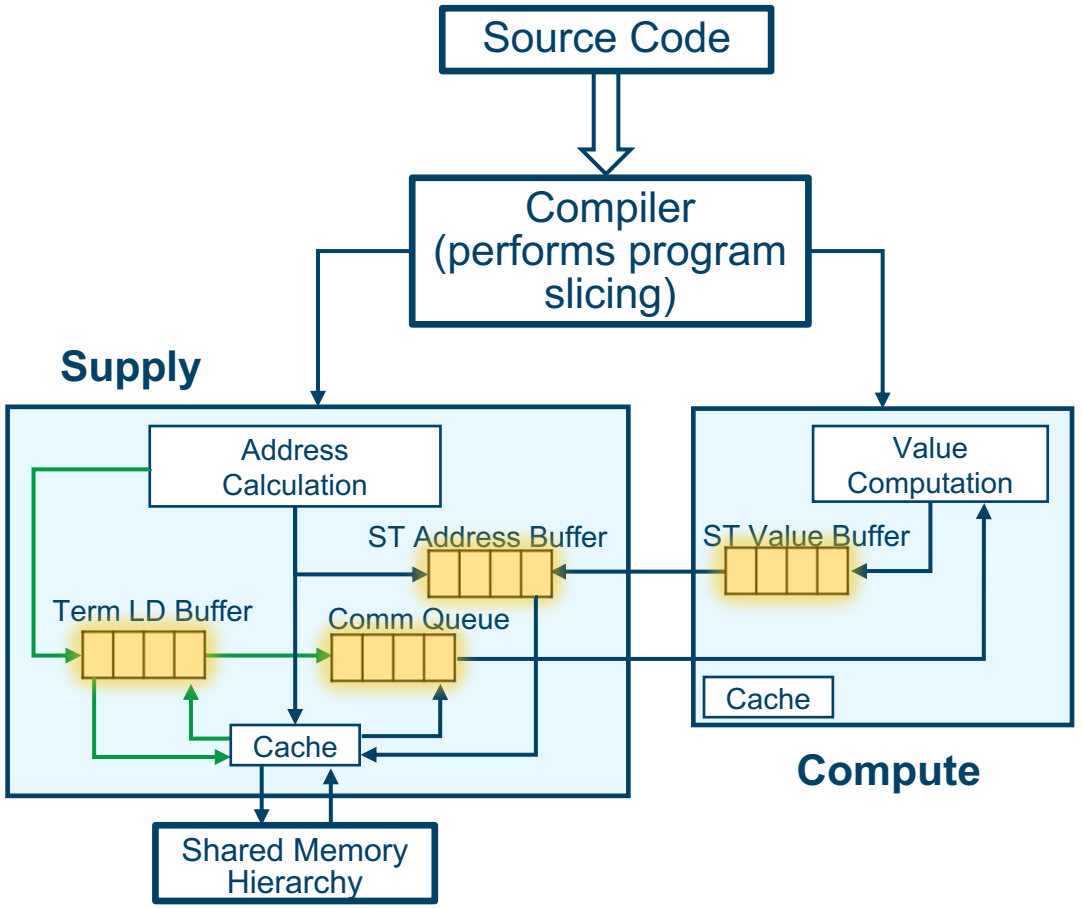


The *Supply* issues several **terminal** loads

[1] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *MICRO*, pg. 191–203. ACM, 2015.

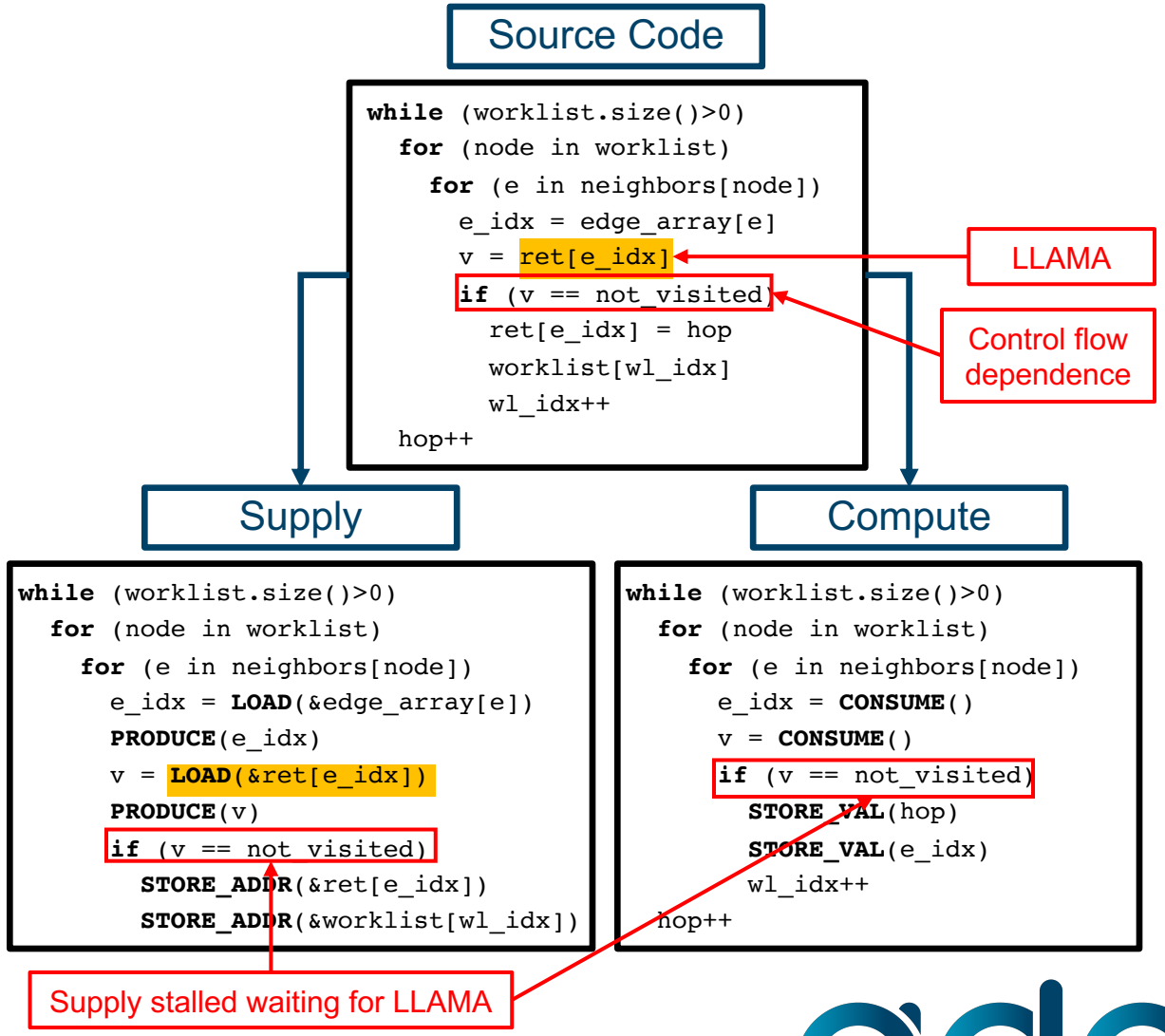
Decoupling from a Hardware Perspective

- **Communication Queue**: holds data from loads the *Supply* has completed
  - Loads completed via a **`PRODUCE`** instruction
  - The *Compute* core's **`CONSUME`** instruction reads and uses this data
- **Store Address Buffer**: holds addresses of **`STORE_ADDR`** instructions whose values have yet to be calculated
- **Store Value Buffer**: holds computed values in case they need to be forwarded to dependent loads
- **Terminal Load Buffer**: holds terminal load memory requests and places them on the communication queue once they are served



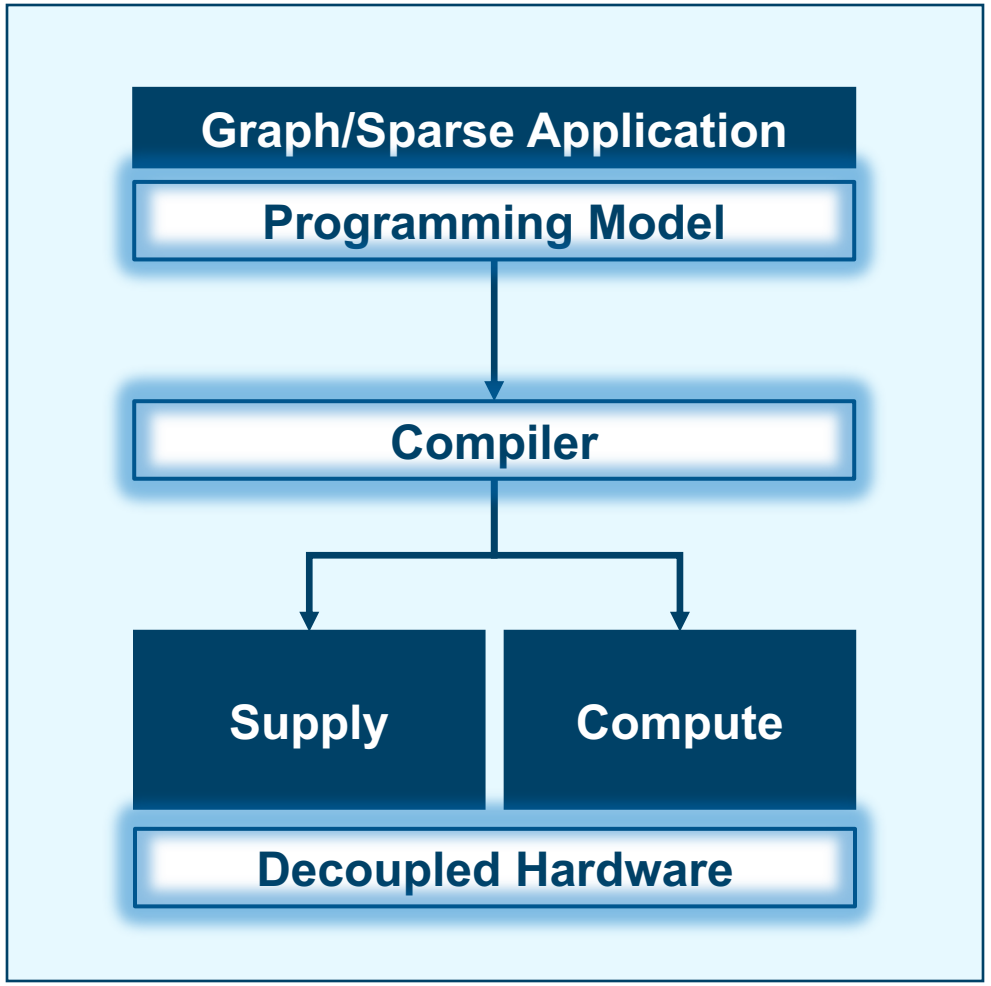
Challenges with Decoupling in Graph Apps

- Graph/sparse applications have small compute to memory ratios
  - Little work to overlap if the *Supply* is stalled
- Control flow dependencies inhibit the *Supply* from running ahead
  - The Supply pipeline is stalled while waiting for data to return from memory
- In many graph apps, there is a control flow dependence on a LLAMA
  - This makes it difficult to achieve MLP with LLAMAs
  - Prior decoupling approaches struggle with such apps



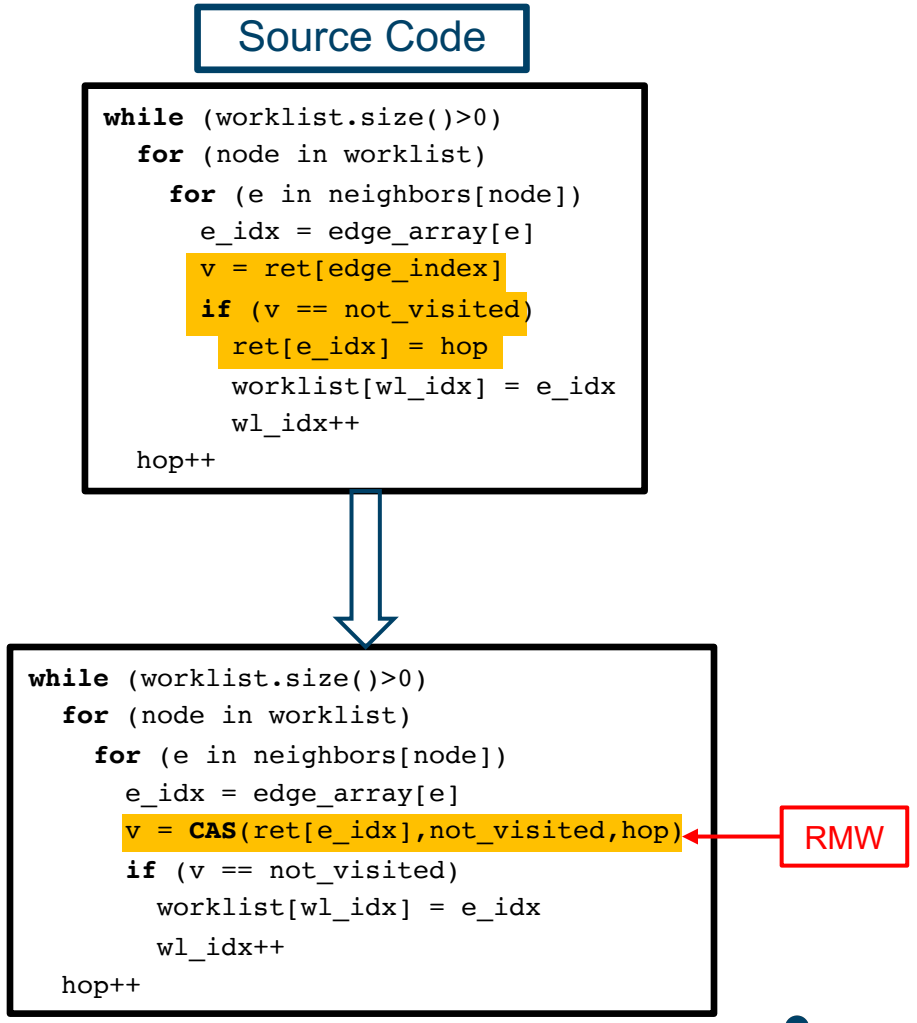
Decoupling for Graph Apps: FAST-LLAMAs

- **FAST-LLAMAs**: Full-stack Approach and Specialization Techniques to address the LLAMAs present in graph/sparse applications
  - The **programming model** allows for explicit decoupling annotations
    - Useful for identification of the LLAMA for transformation into a terminal access
  - The **compiler** performs program slicing, sometimes more aggressively with annotations
    - Identifies terminal access opportunities
  - The **decoupled hardware** efficiently performs long-latency memory accesses to exploit MLP
    - In-order *Supply* and *Compute* cores perform decoupling



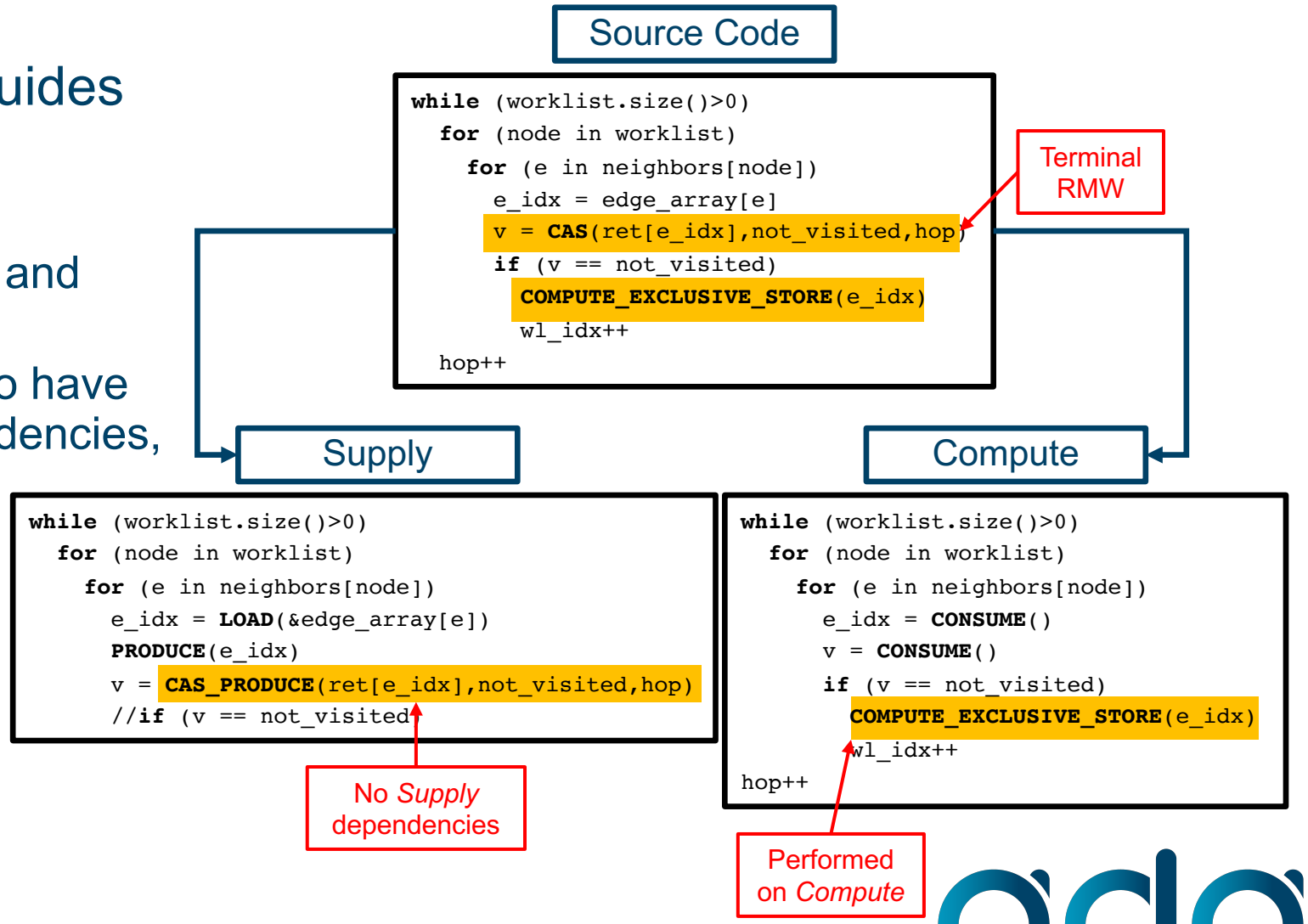
Transforming the LLAMA

- Expressions in the source code of these graph/sparse applications can be transformed into **read-modify-write (RMW)** operations
  - RMWs atomically read data, modify it based on a condition, and write the modified data
  - Allows parallelism over node processing
- VP graph applications mimic RMW operation behavior for neighbor processing
  - Ex: In BFS, change node data if its value is -1 (node is not visited)
  - Compare And Swap example:
    `CAS``(addr,to_compare,val)`
    - Atomically compares the value stored at `addr` to `to_compare`; if they are equal, stores `val` at `addr`



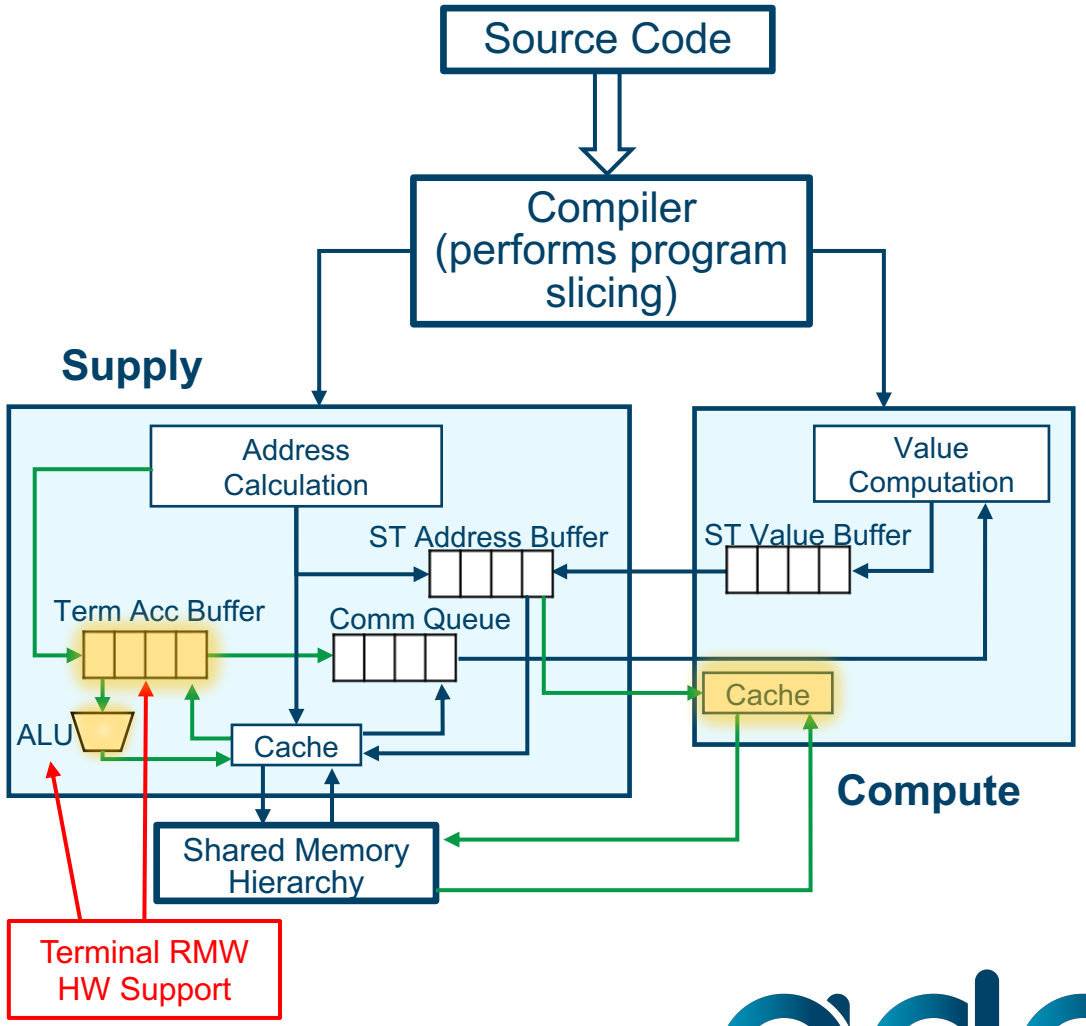
Making RMWs Terminal

- **Compute-exclusive store**: programmer annotation that guides the compiler to place memory accesses on the *Compute*
  - Removes store from the *Supply* and leaves it on the *Compute*
  - Allows the access in the RMW to have no *Supply* or control flow dependencies, creating a **terminal RMW**
- **LLAMAs** are effectively transformed into **terminal RMWs**
  - The *Supply* can now runahead and issue these early to hide memory latency

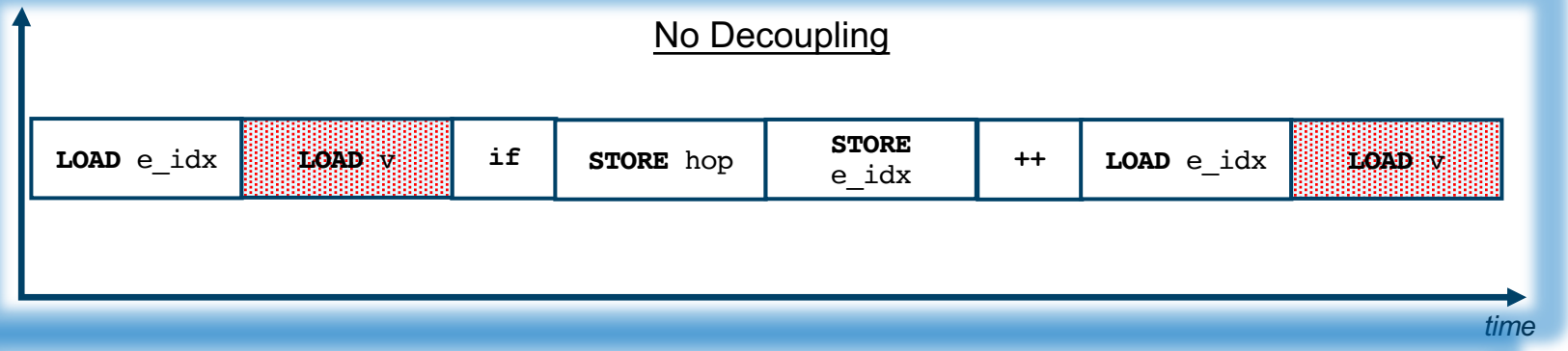


FAST-LLAMAs Decoupled Hardware

- **Terminal load buffer** generalized to a **terminal access buffer**
  - Can support both **terminal loads** and **terminal RMW accesses** (“read”)
- Addition of an ALU on the **terminal access buffer** to perform “modify” operations (e.g. compare, add)
- **Terminal RMW** sends out a request for data from memory
  - Requires atomicity with memory operations
  - Memory system guarantees exclusive permissions to data’s cacheline with a lock
- “Read” result enqueued and atomic store (“write”) sent to the memory hierarchy
- Use of the *Compute* cache



FAST-LLAMAs Mitigates Latency Effects



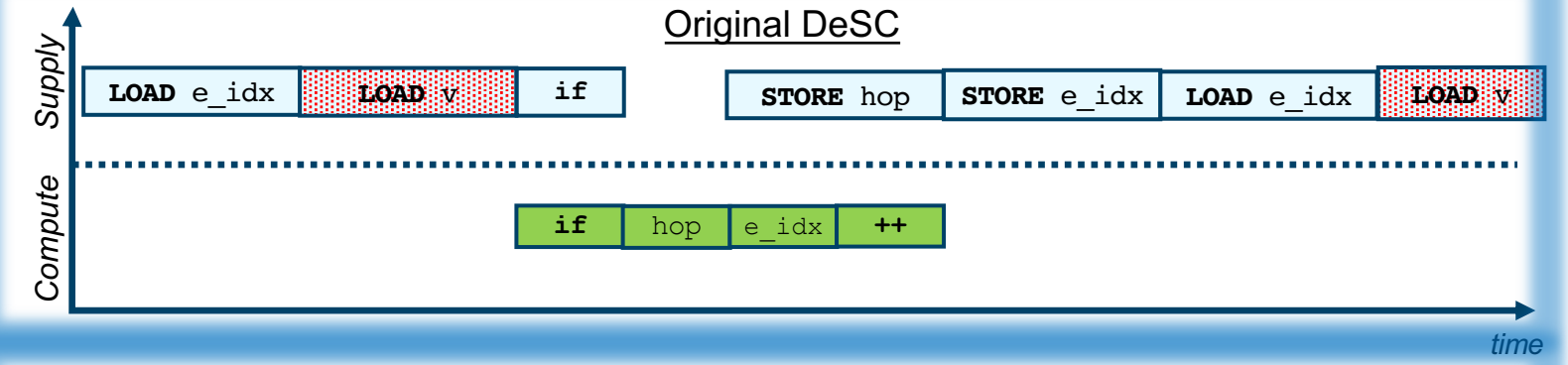

```
for (e in neighbors[node])
  e_idx = edge_array[e]
  v = ret[edge_index]
  if (v == not_visited)
    ret[e_idx] = hop
    worklist[wl_idx] = e_idx
    wl_idx++
```

operation that maps to Supply

async operation that uses terminal RMW

LLAMA

operation that maps to Compute

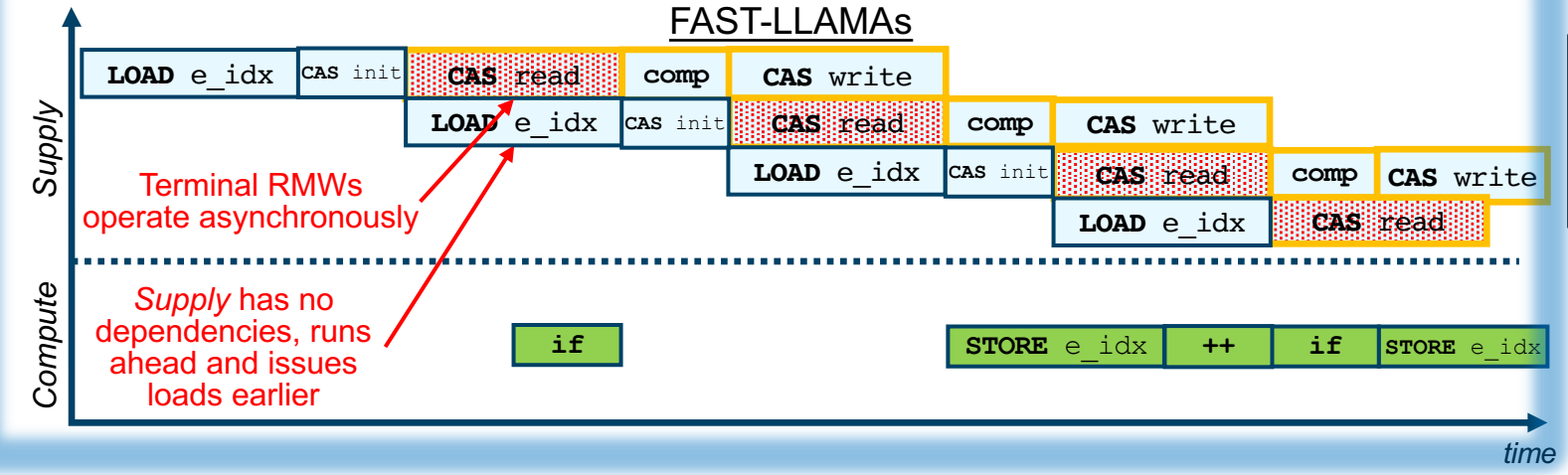


Supply

```
for (e in neighbors[node])
  e_idx = LOAD(&edge_array[e])
  PRODUCE(e_idx)
  v = LOAD(&ret[e_idx])
  PRODUCE(v)
  if (v == not_visited)
    STORE_ADDR(&ret[e_idx])
    STORE_ADDR(&worklist[wl_idx])
```

Compute

```
for (e in neighbors[node])
  e_idx = CONSUME()
  v = CONSUME()
  if (v == not_visited)
    STORE_VAL(hop)
    STORE_VAL(e_idx)
    wl_idx++
```



Supply

```
for (e in neighbors[node])
  e_idx = LOAD(&edge_array[e])
  PRODUCE(e_idx)
  v = CAS(ret[e_idx],
      not_visited,hop)
```

Compute

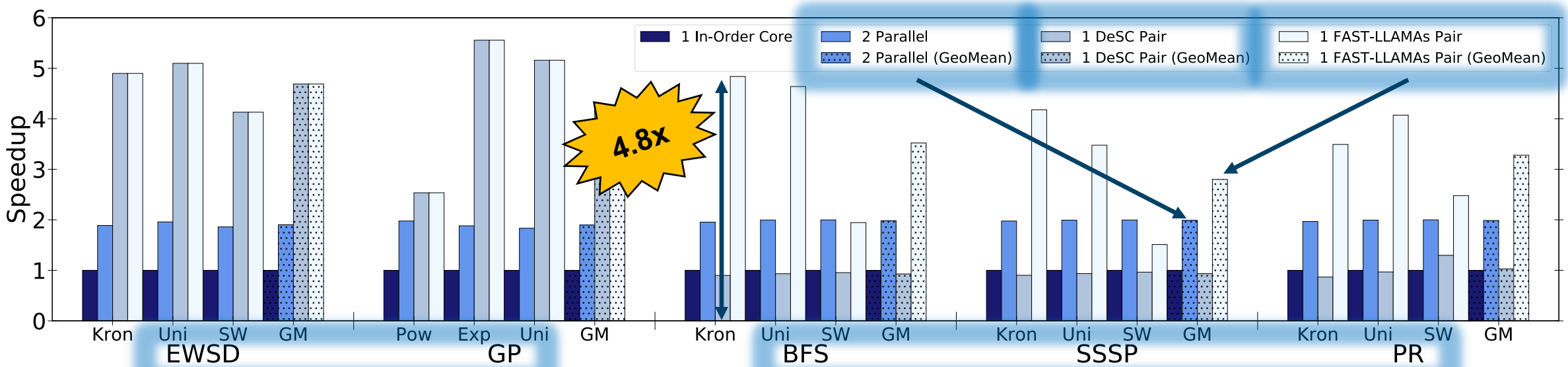
```
for (e in neighbors[node])
  e_idx = CONSUME()
  v = CONSUME()
  if (v == not_visited)
    COMPUTE_EXCLUSIVE_STORE(e_idx)
    wl_idx++
```

How do these approaches perform?

2 Parallel In-Order Cores

1 In-Order DeSC Pair

1 In-Order FAST-LLAMAs Pair



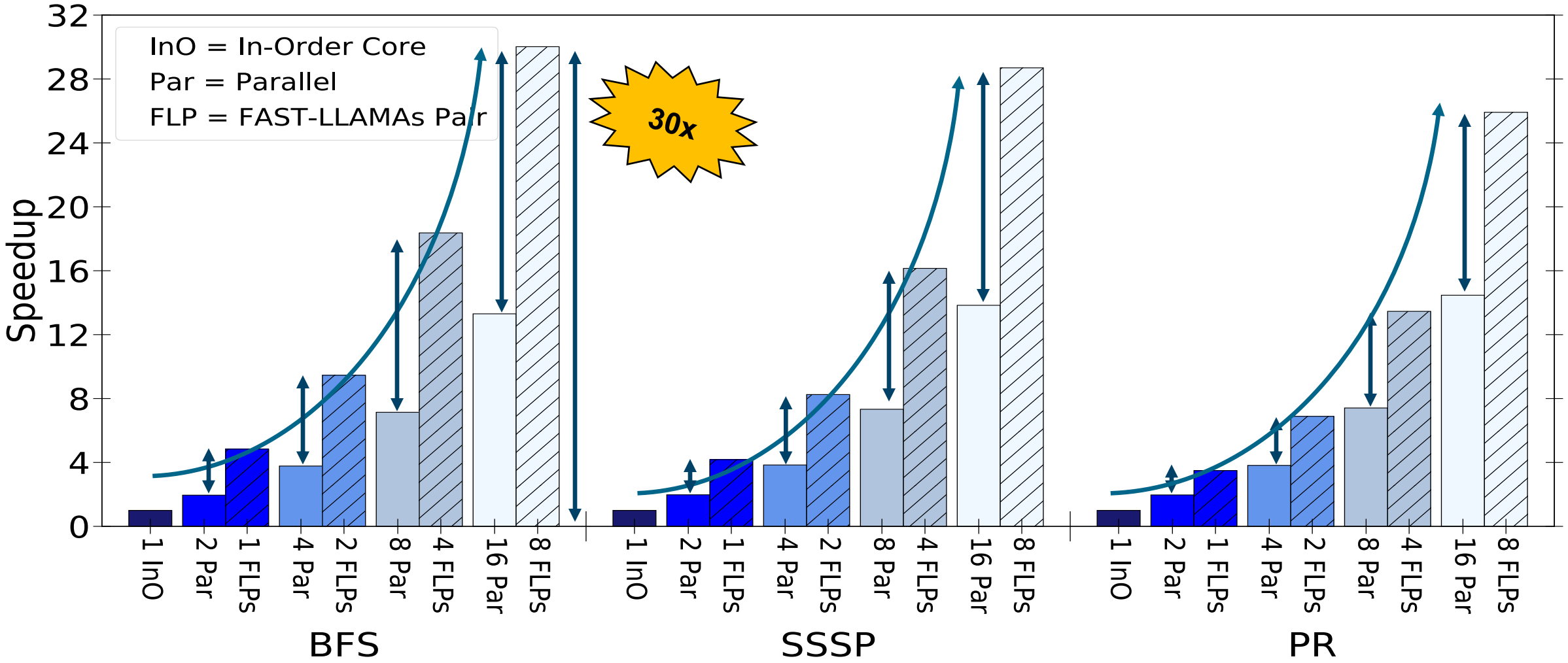
Apps where LLAMA is naturally terminal load

Apps where LLAMA has a control flow dependence:
Vertex Programmable (VP) apps

PI: Martonosi

Applications Driving Architectures

How well do these approaches scale?

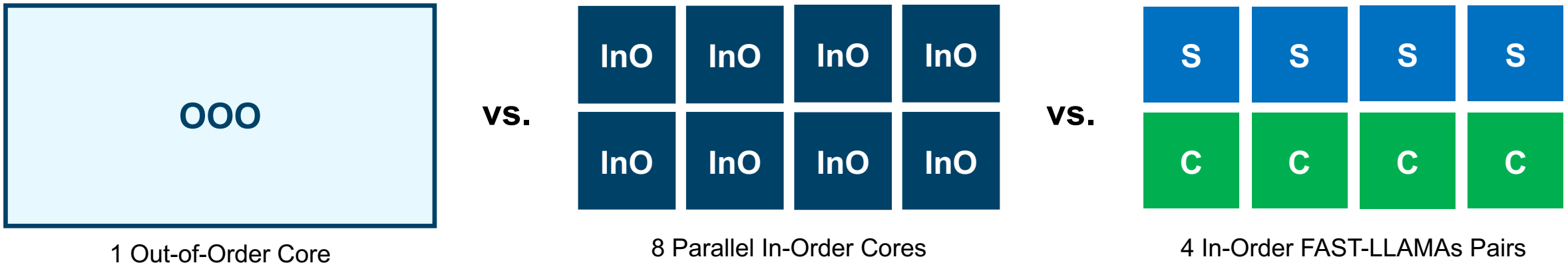


FAST-LLAMAs scales **superlinearly** with respect to the number of cores!

PI: Martonosi

Applications Driving Architectures

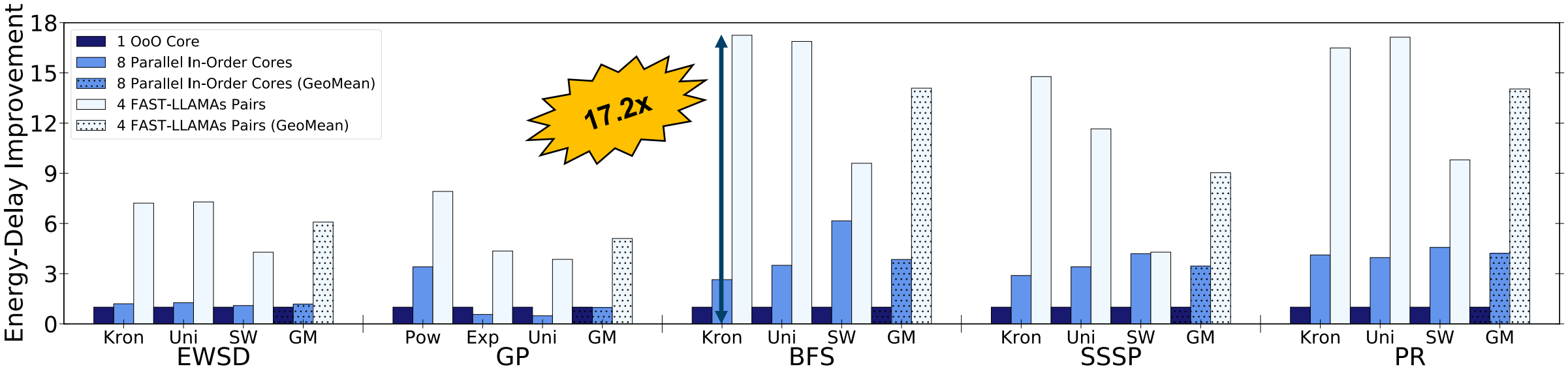
How energy efficient are they?



1 Out-of-Order Core

8 Parallel In-Order Cores

4 In-Order FAST-LLAMAs Pairs



Conclusion

- Graph/sparse applications are characterized by LLAMAs whose latencies dominate the total runtimes
- Thus, modern technology trends have struggled to accelerate such applications
  - Out-of-order cores, traditional parallelism, and prior decoupling techniques have been utilized to tolerate memory latency
  - These techniques perform with varying degrees of success on some graph/sparse applications, but fall short on many important kernels
- Full-stack innovations are necessary, particularly for graph/sparse applications
- FAST-LLAMAs successfully tolerates memory latency by revisiting decoupling and proposing an innovative **terminal RMW** operation
  - Achieves a **4.8x** speedup over a single InO and up to **3x** better than traditional parallelism
  - Scales performance improvements to **30x** with 16 cores
  - Yields a **17.2x** improvement in energy efficiency over OoO

Questions

- What are some examples of big data applications you have used or encountered in your institution/company that employ graph analytics?
- How frequently do you employ GPUs to accelerate these applications?
- What are your thoughts on using in-order cores with full-stack decoupling support as opposed to mapping graph applications onto GPUs?

JUMP

Joint University Microelectronics Program

www.src.org/program/jump

Semiconductor Research Corporation

@srcJUMP

Applications Driving Architectures