http://inst.eecs.berkeley.edu/~ee290-2/sp21/

Hardware for Machine Learning

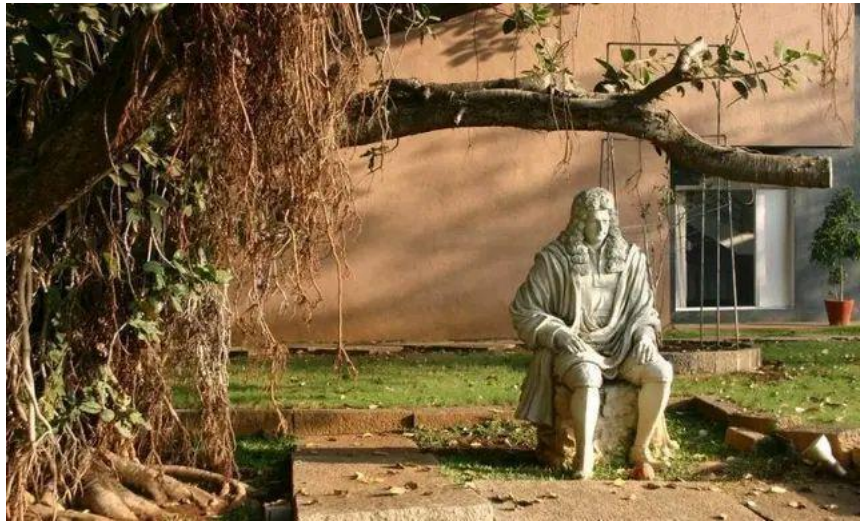
Lecture 12: Data Orchestration

Sophia Shao

Stay Calm. Stay Creative: Newton and the Great Plague

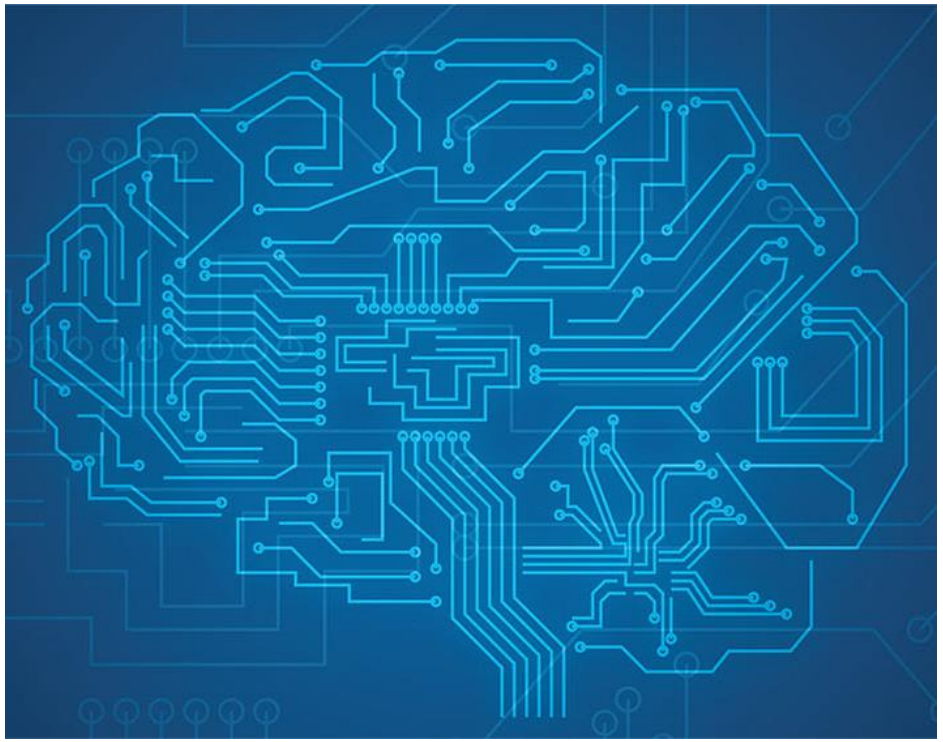
"In 1665, following an outbreak of the bubonic plague in England, Cambridge University closed its doors, forcing Newton to return home to Woolsthorpe Manor. While sitting in the garden there one day, he saw an apple fall from a tree, providing him with the inspiration to eventually formulate his law of universal gravitation. Newton later relayed the apple story to William Stukeley, who included it in a book, "Memoir of Sir Isaac Newton's Life," published in 1752."

https://www.history.com/news/9-things-you-may-not-know-about-isaac-newton



Review

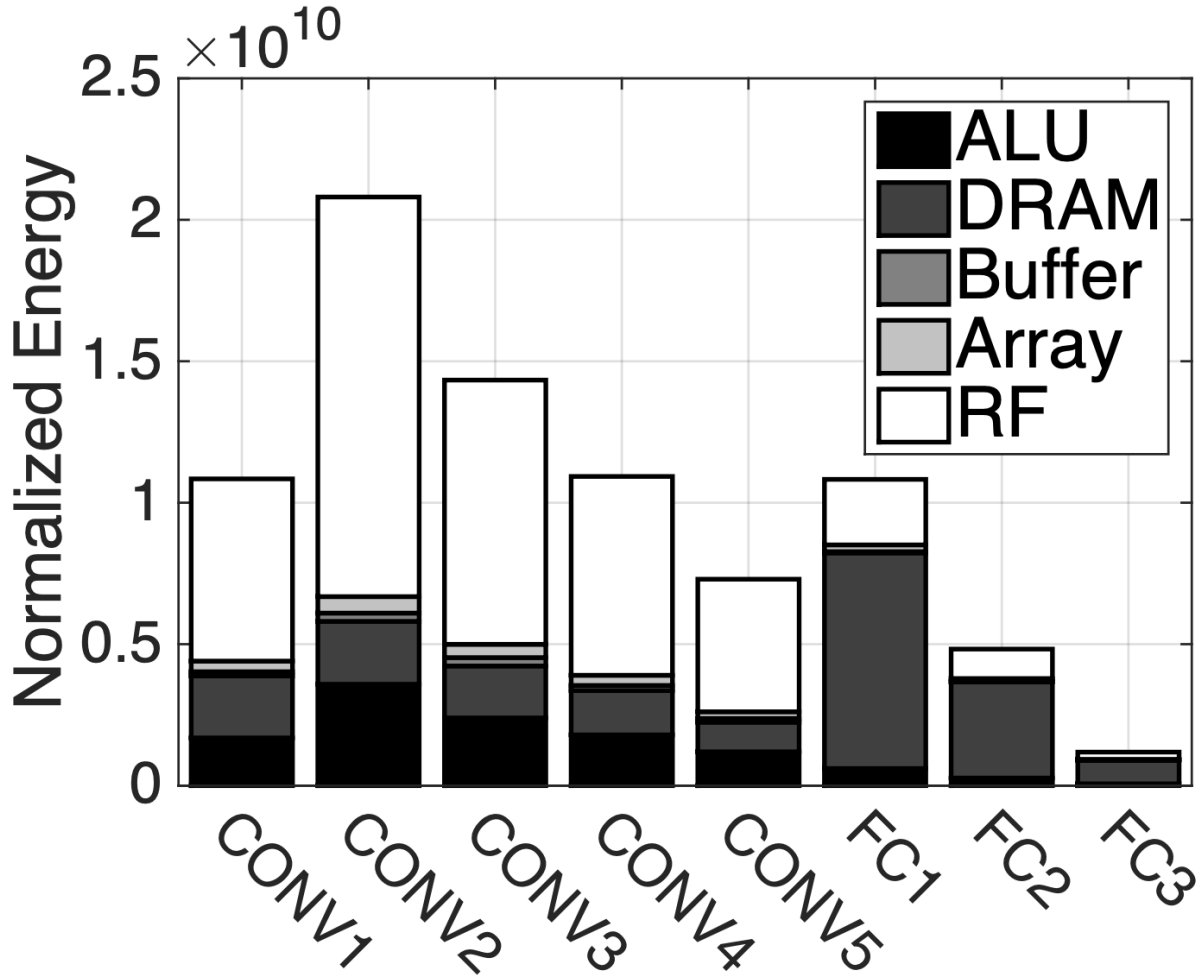
- Core computation in DNN
- Execution order of the core computation
- Hardware realization of the core computation
- Mapping DNNs to hardware
  - Temporal and Spatial Mapping based on hardware constraints.
  - Tile the loops to improve reuse and parallelism
  - Navigate the large mapping space
- This lecture: data transfer mechanisms across storage hierarchy



Data Orchestration

- Overview
  - Design Principles
- Taxonomy:
  - Implicit vs Explicit
  - Coupled vs Decoupled
- Case studies

Data movement dominates ML HW energy.



Eyeriss, ISCA'2016

Buffer hierarchy in ML accelerators.

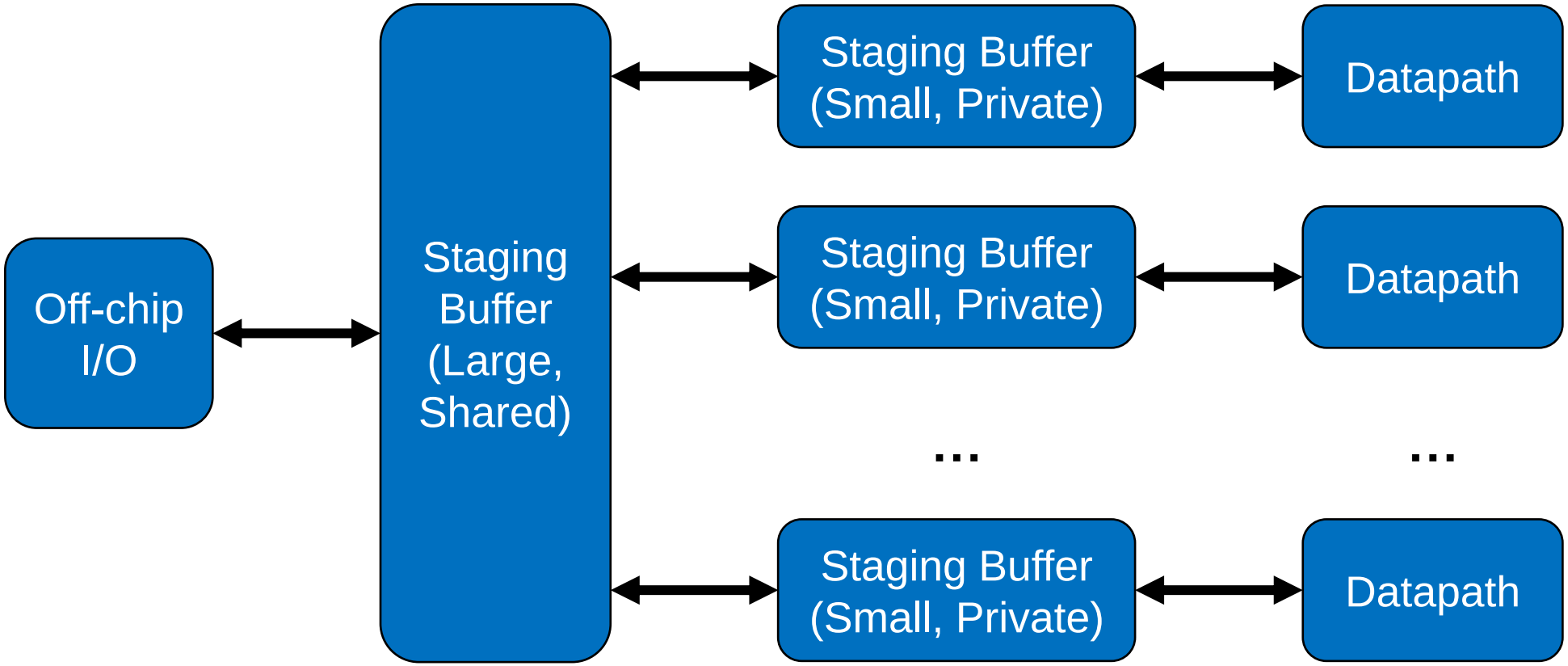
- Percentage of on-chip area that devotes to on-chip buffers:

DaDianNao [5]:	48%	Eyeriss [6]:	40%-93%
EIE [18]:	93%	SCNN [35]:	57%
TPU [22]	35%	PuDianNao [27]	63%

Buffets, ASPLOS'2019

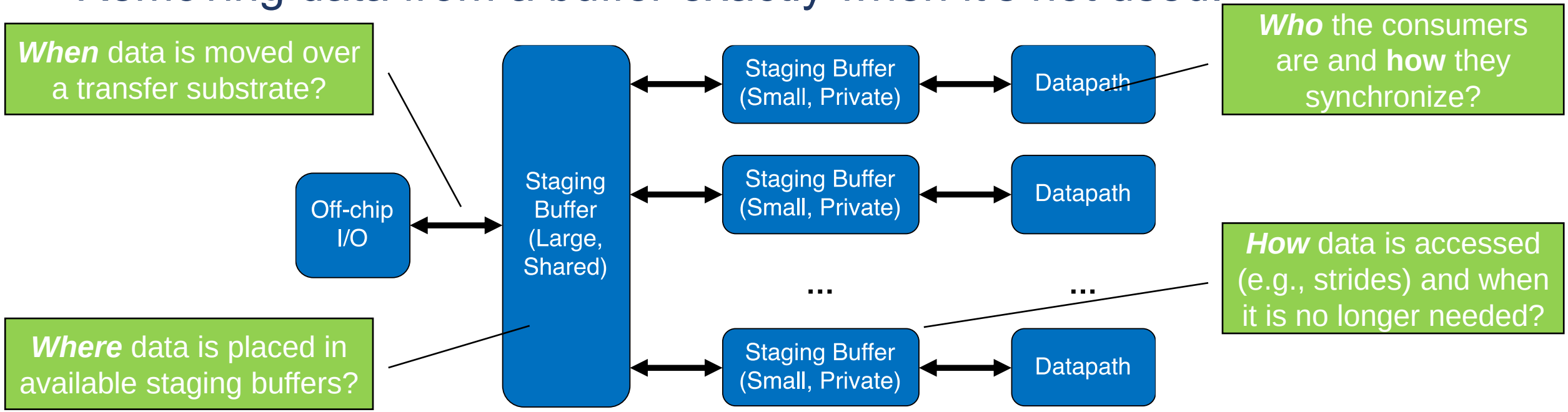
What's data orchestration?

- Feeding data to a functional unit exactly when it wants it.
- Removing data from a buffer exactly when it's not used.



What's data orchestration?

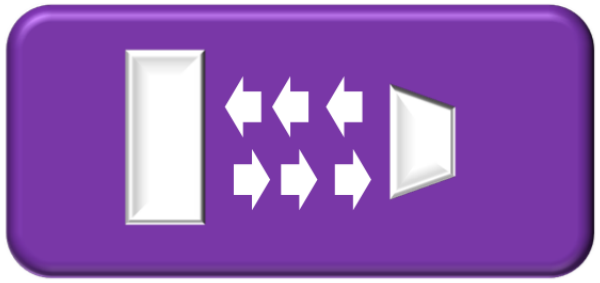
- Feeding data to a functional unit exactly when it wants it.
- Removing data from a buffer exactly when it's not used.



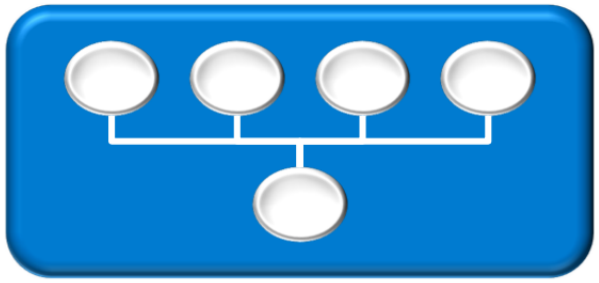
- ML accelerators use workload knowledge to optimize data orchestration at design time.

Buffets, ASPLOS'2019

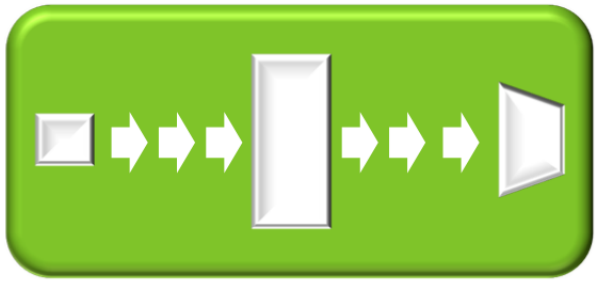
Guiding Principles



Local reuse – staged physically close to consuming units

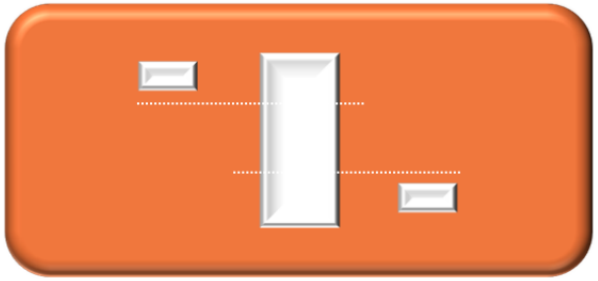


Cross-unit use – amortize data access and communication



Bandwidth efficiency - Maximize delivery rate by controlling outstanding requests

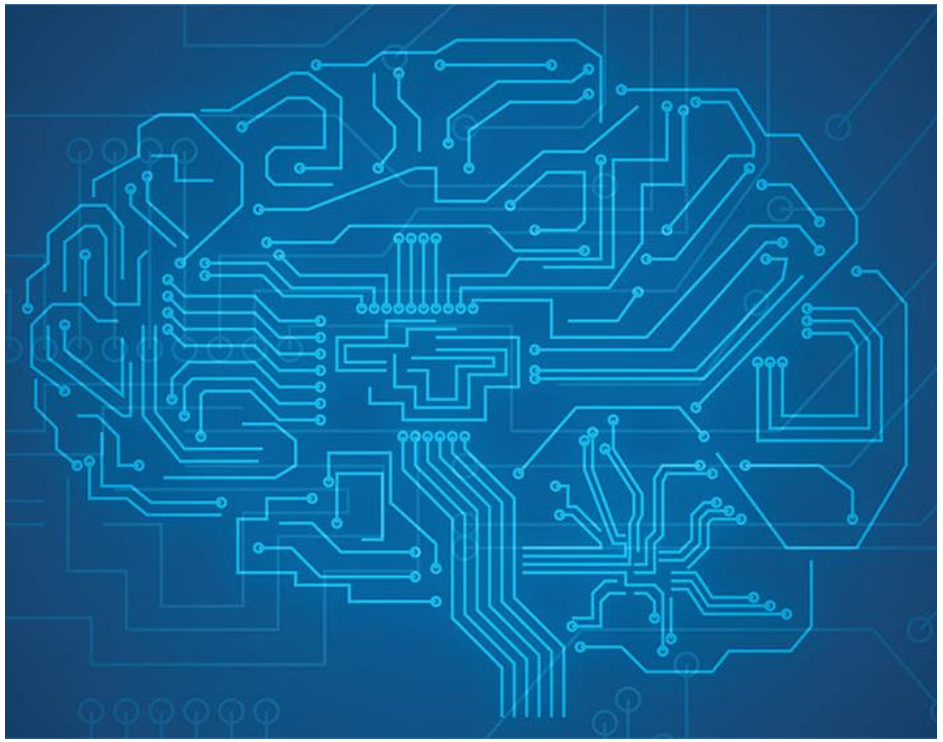
Delivery/use overlap - Next tile should be available when current is done (e.g., double-buffering)



Precise synchronization – Only wait for exactly data you need, respond quickly (e.g., no barriers or remote polling)

Simple structures - Minimize hardware area/power

Buffets, ASPLOS'2019

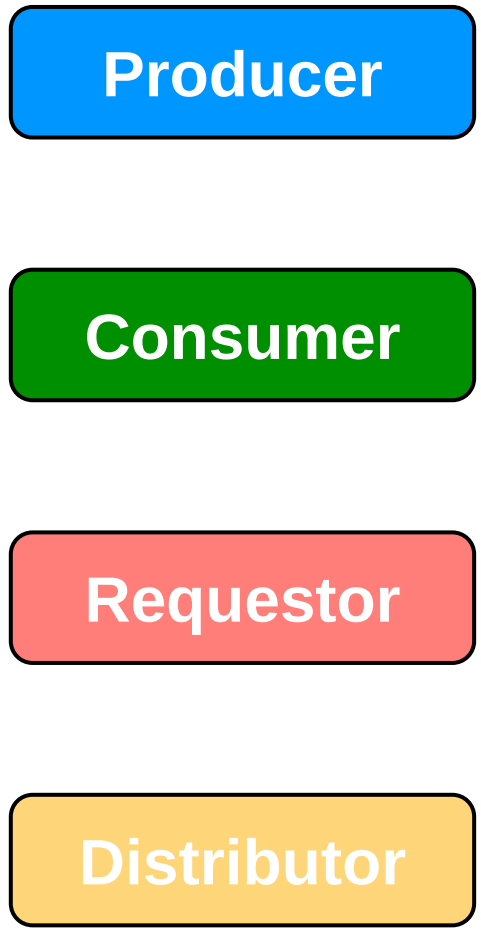


Data Orchestration

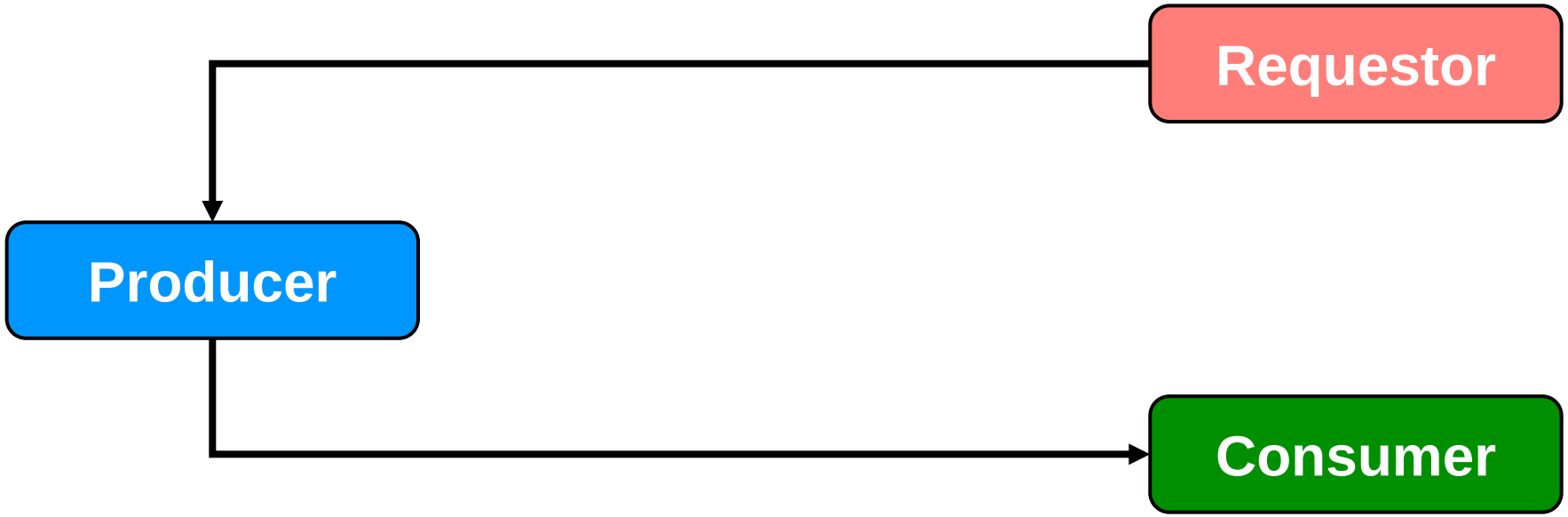
- Overview
  - Design Principles
- Taxonomy:
  - Implicit vs Explicit
  - Coupled vs Decoupled
- Case studies

Agents in Data Orchestration

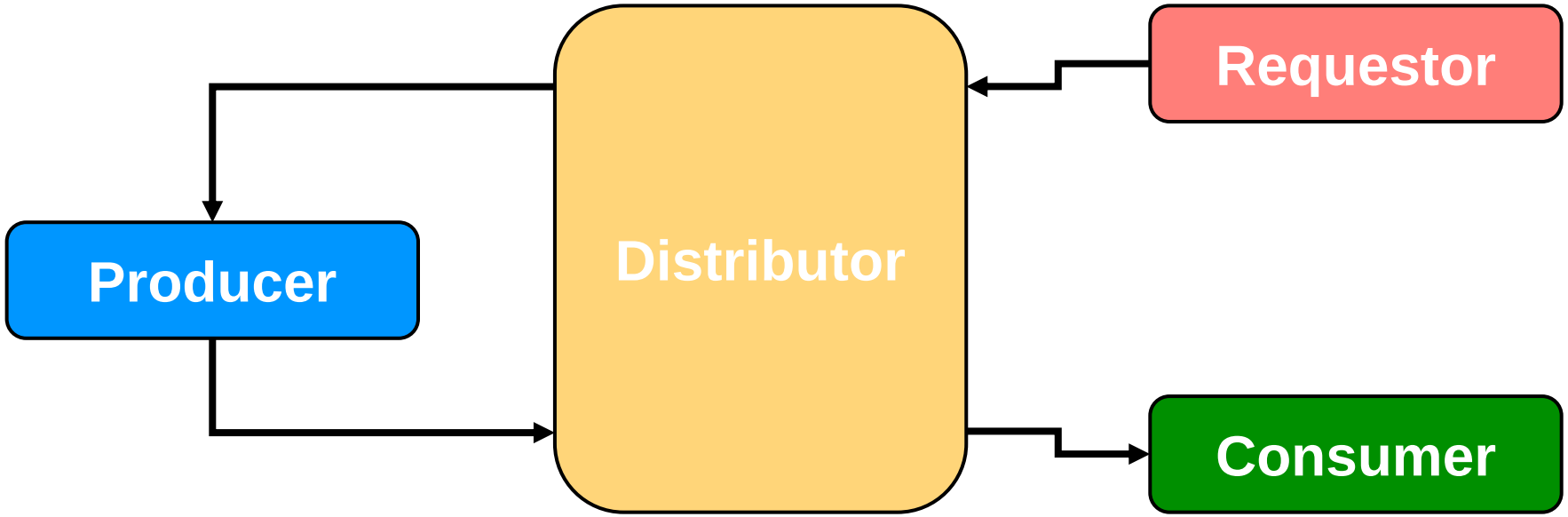
- Data Producer:
  - The agent that currently contains the requested data.
- Data Consumer
  - The agent that consumes the requested data.
- Data Requestor
  - The agent that sends out data requests.
- Data Distributor
  - The agent that distributes data requests.



Agents in Data Orchestration



Agents in Data Orchestration

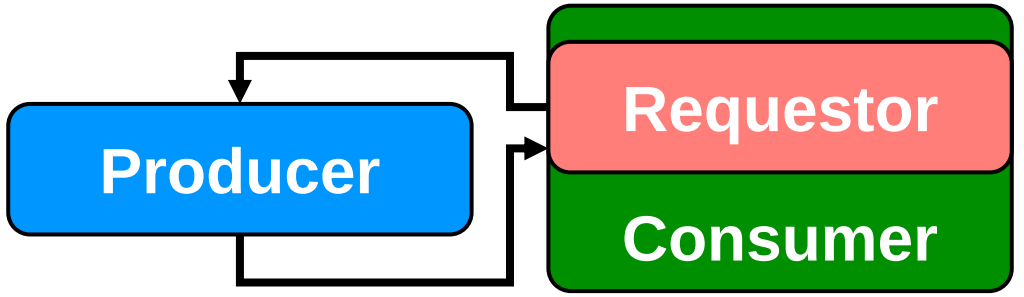


Coupled vs. Decoupled

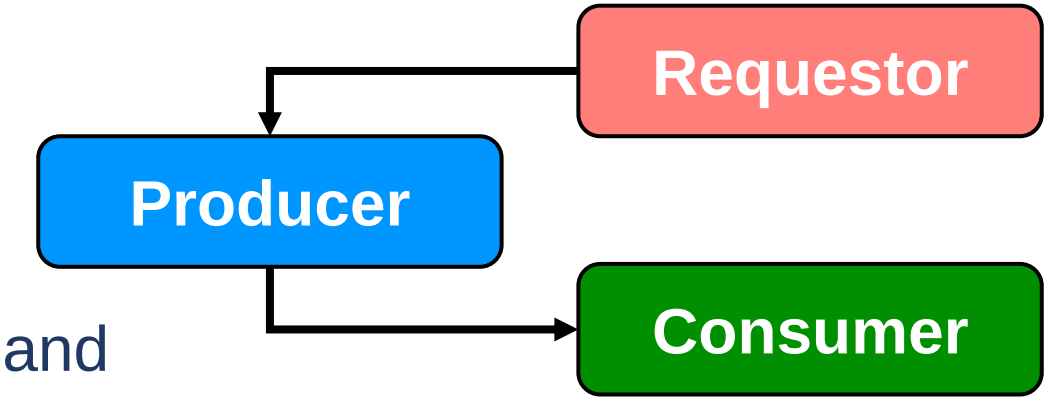
- Whether requestor == consumer.

- Coupled
  - Requestor is the same as Consumer.
  - Pro: Easy and intuitive synchronization.
  - Con: Reserved landing zone (e.g., registers) for incoming data.

- Decoupled
  - Requestor is different from Consumer.
  - Pro: Requestor can run ahead.
  - Con: Synchronization between requestor and consumer.



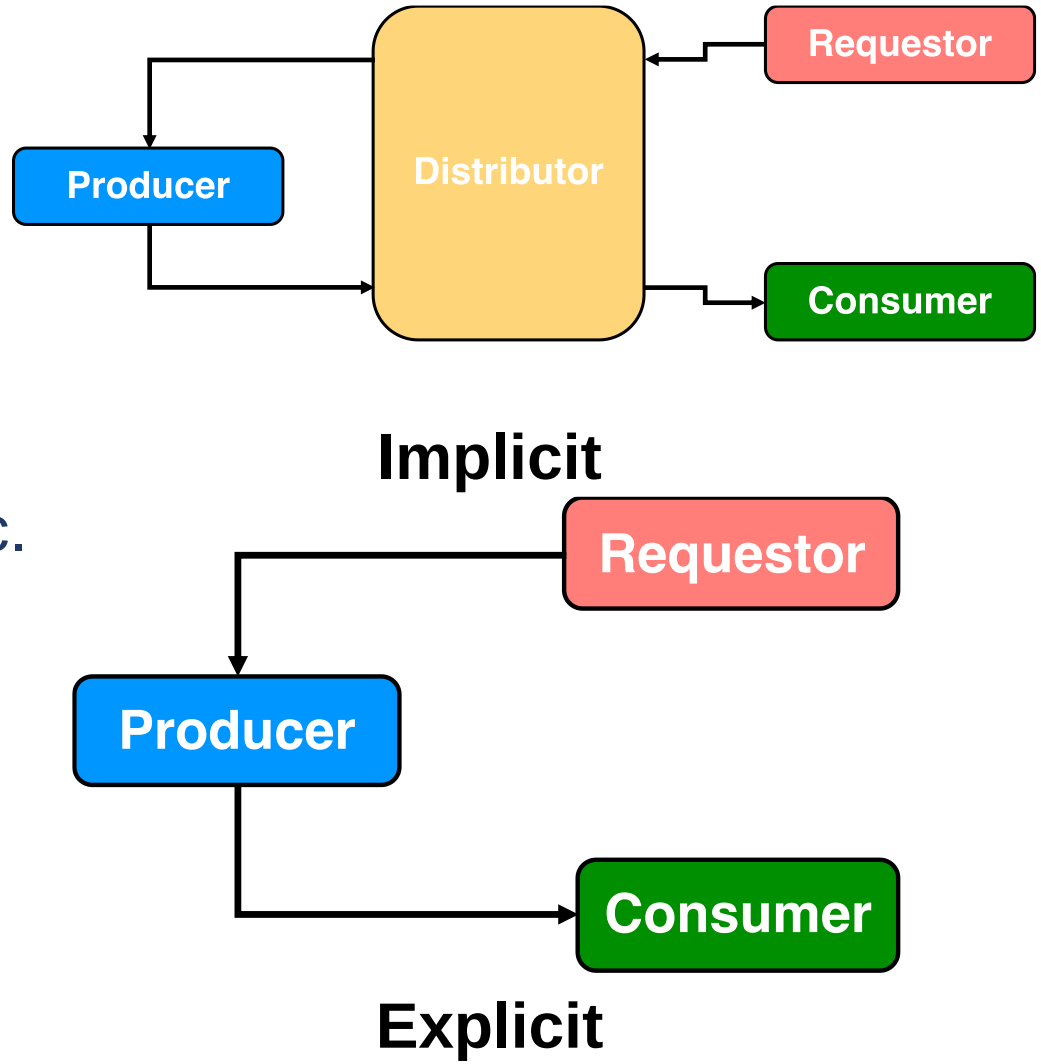
Coupled



Decoupled

Implicit vs. Explicit

- Whether requests are sent to data producer directly.
- Implicit:
  - Requestor is not aware of where the data is and when the data is evicted.
  - Pro: Easy to program; Workload agonistic.
  - Con: Area and energy overhead for tags etc.
- Explicit:
  - Requestor explicitly interacts with producer.
  - Pro: Low overhead.
  - Con: Hard to program.



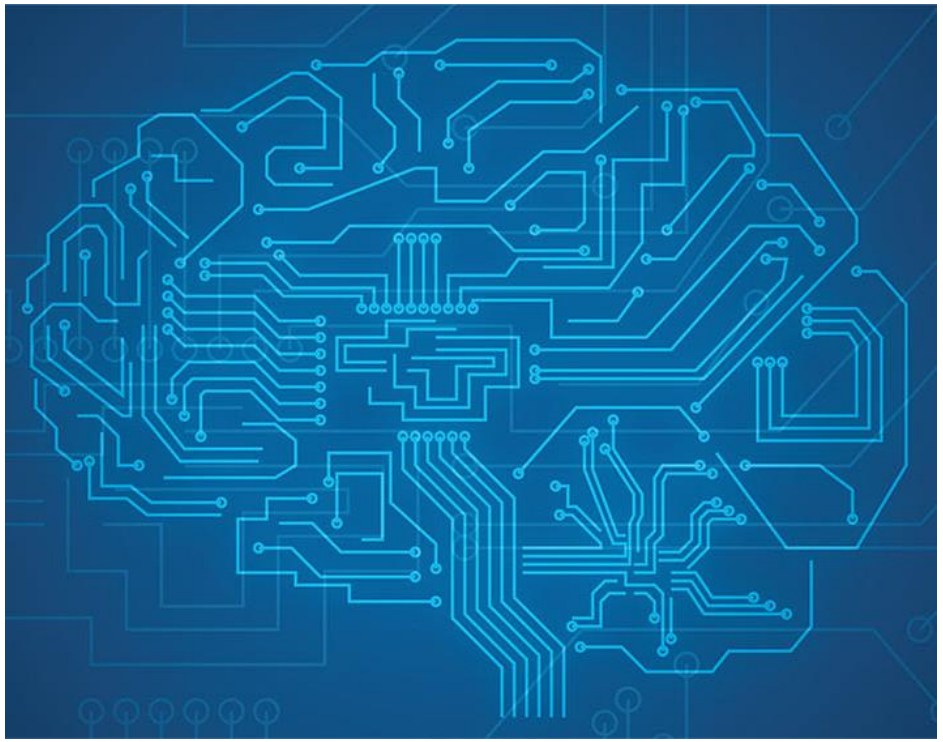
Data Orchestration Taxonomy

	Coupled	Decoupled
Implicit		
Explicit		

Administrivia

- Lab 3 is due this week.
  - Have fun!
  - Deadline extended to Sunday.

- Project starts next week:
  - Project meet-up last & this weeks.
    - 3/4 2-3pm
  - Talk to the teaching staff about your project ideas/scope.
  - 3/19 Project Proposal Due



Data Orchestration

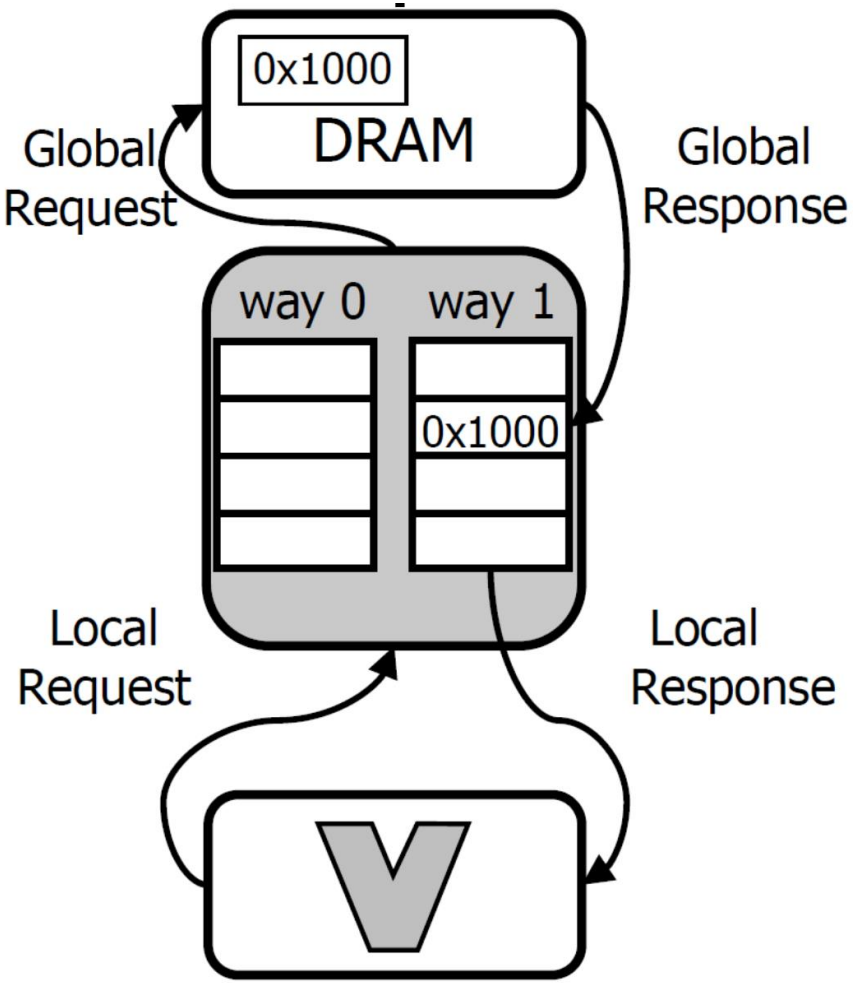
- Overview
  - Design Principles
- Taxonomy:
  - Implicit vs Explicit
  - Coupled vs Decoupled
- Case studies

Data Orchestration Taxonomy

	Coupled	Decoupled
Implicit	Cache	Decoupled Access-Execute
Explicit	GPU shared memory	DMA

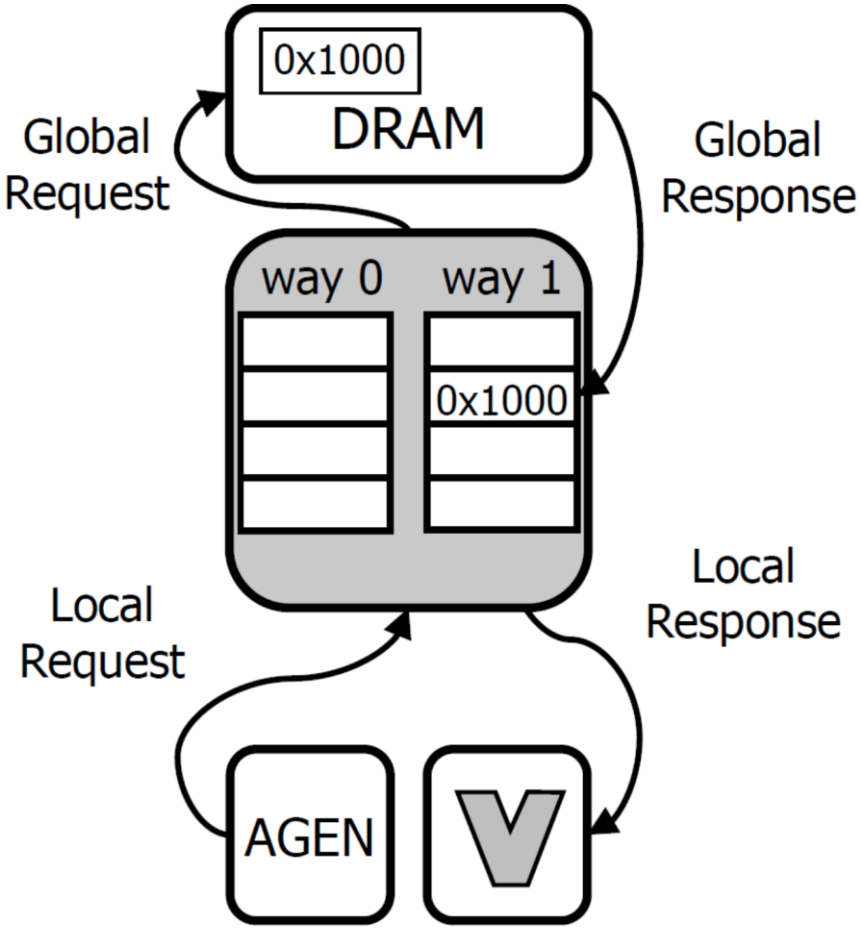
Implicit-Coupled Data Orchestration

- Cache
  - Widely used in general-purpose computing
  - Reusable, composable
  - High area and energy overhead
- Implicit:
  - Requestor, i.e., core, sends requests to L1 without knowing the exact location of data.
- Coupled:
  - Unified pipeline for mem and compute instructions.



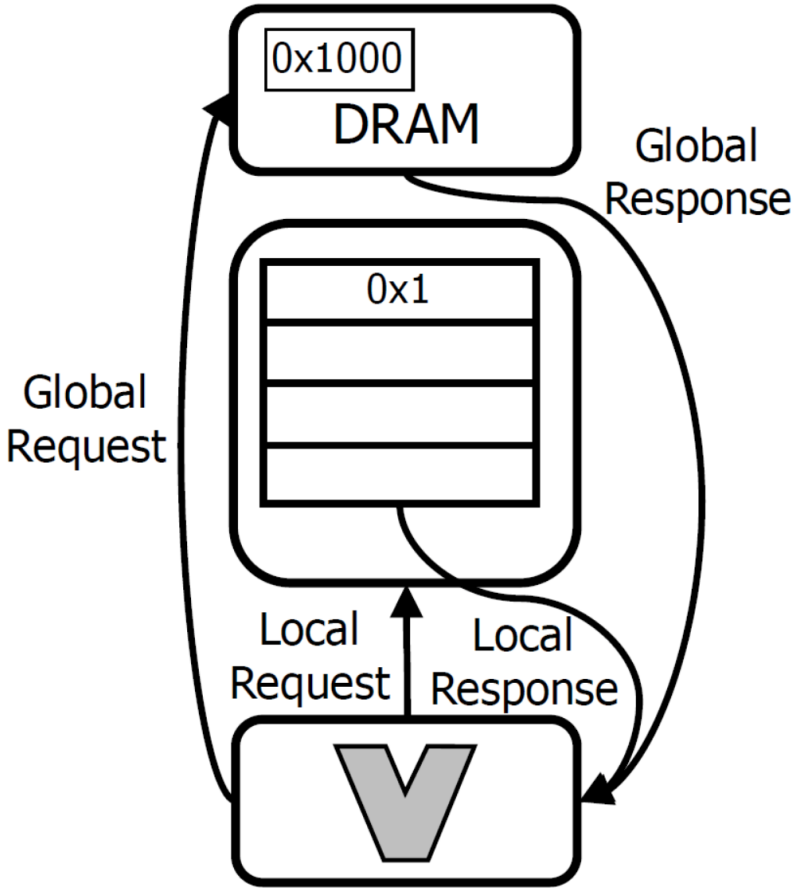
Implicit-Decoupled Data Orchestration

- Decoupled Access-Execute
  - Separating data requestor and consumer connected via hardware queues.
  - Targeting general-purpose computing.
  - Allowing both to proceed at their own rates.
- Implicit:
  - Requestor, or accessor, sends requests to L1 without knowing the exact location of data.
- Decoupled:
  - Separate requestor/accessor and consumer/executor



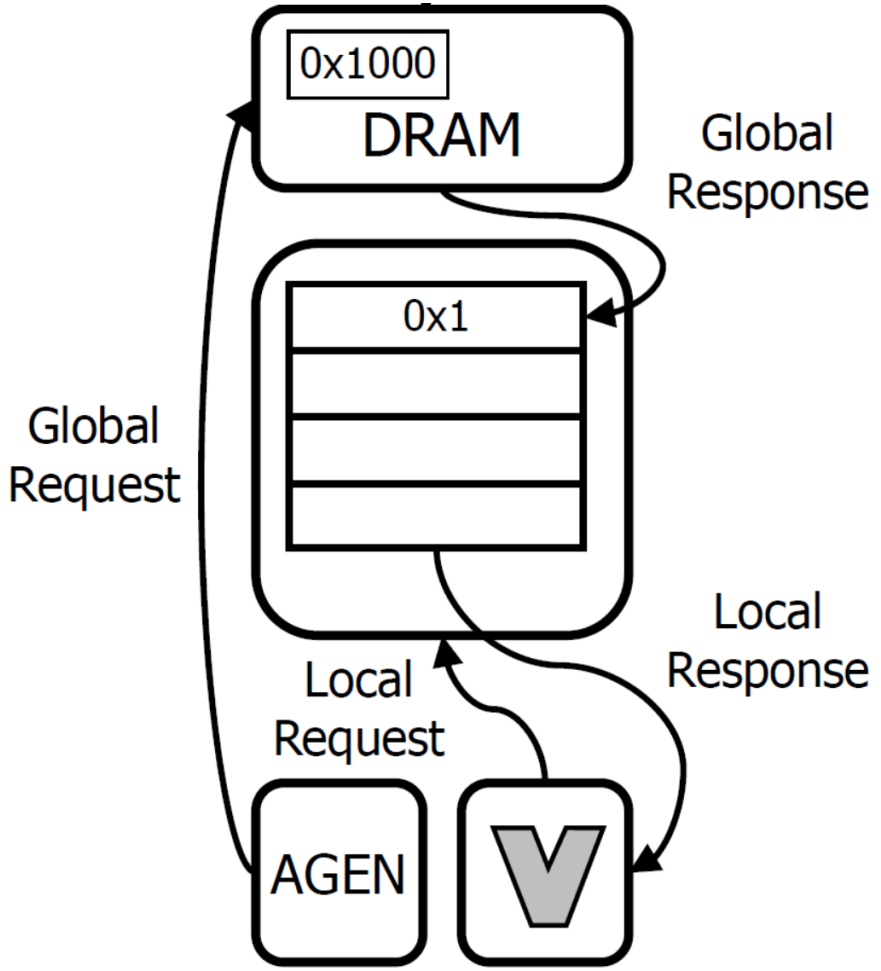
Explicit-Coupled Data Orchestration

- GPU Shared Memory
  - Scratchpad in GPU
  - Instructions to explicitly move data across memory hierarchy
- Explicit:
  - Core explicitly requests data from DRAM
- Coupled:
  - The same pipeline for data requests and consumption.



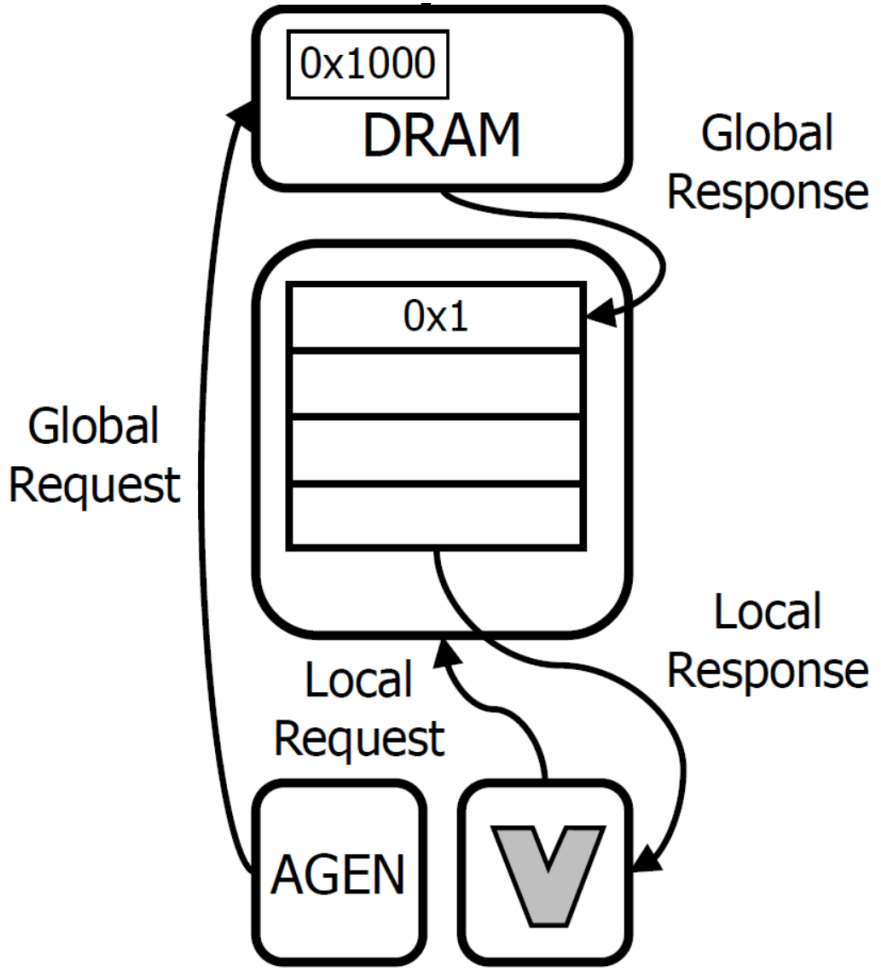
Explicit-Decoupled Data Orchestration

- EDDO
- DMA engines
  - Direct-Memory Access
  - Acts as an address (request) generator to DRAM
  - “Pushing” data to consumer
- Explicit:
  - Software-managed data placement
- Decoupled:
  - DMA only does data orchestration not compute.



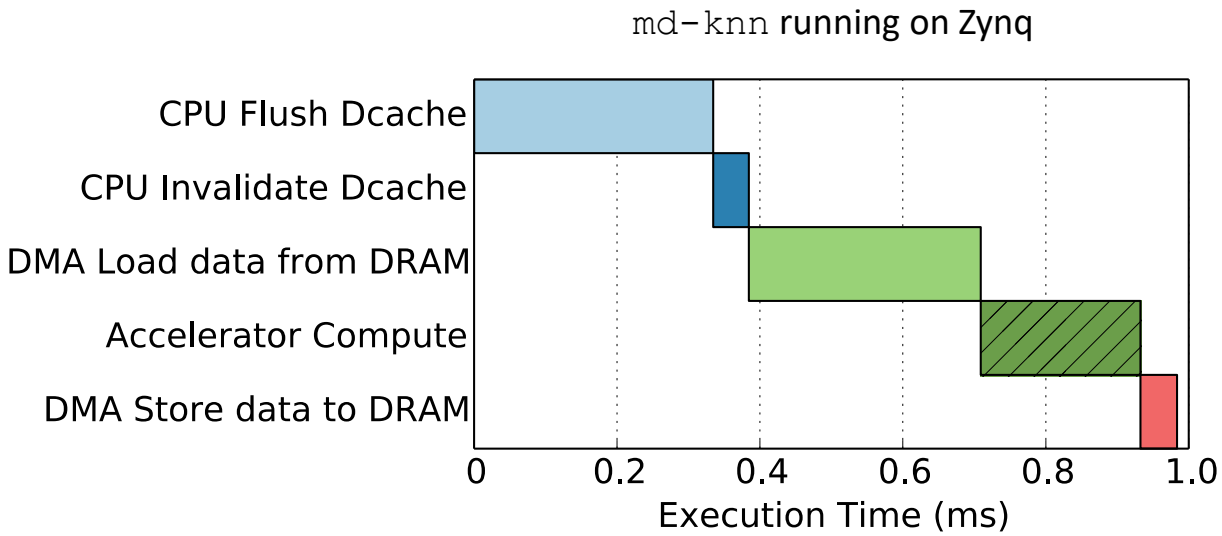
EDDO for Domain-Specific Accelerators

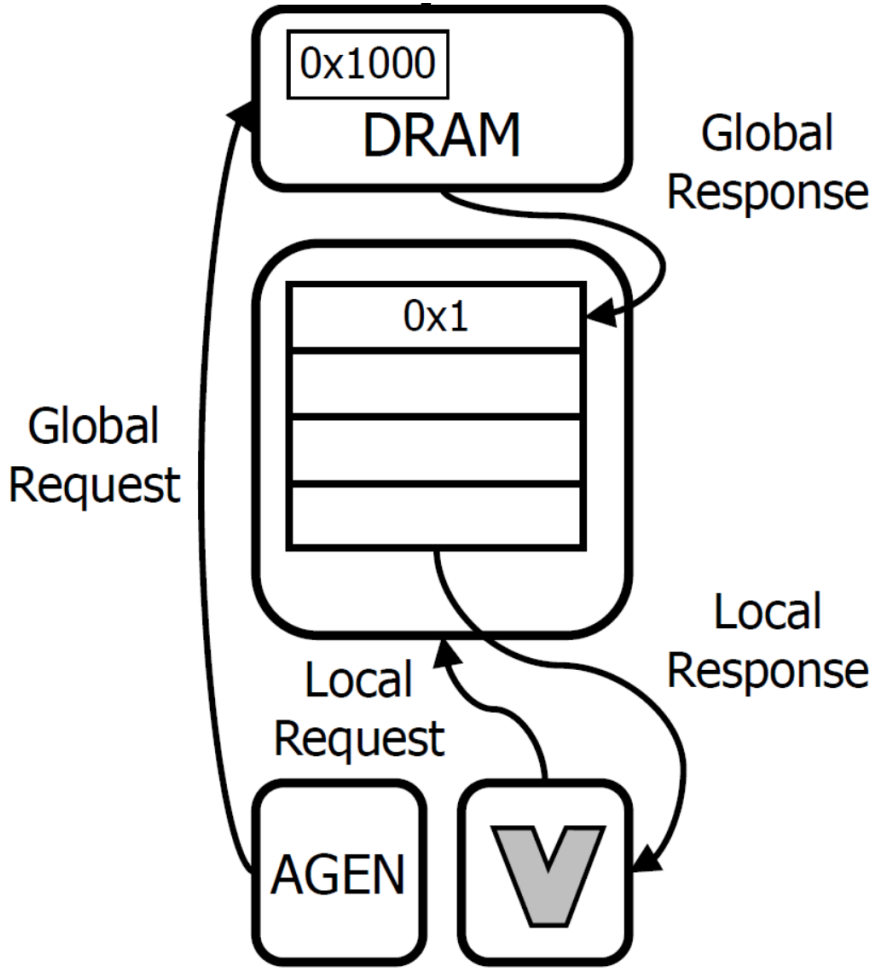
- Benefits:
  - Leverage static workload knowledge
  - Efficient hardware implementations
- Challenges:
  - Synchronization in a decoupled system



EDDO for Domain-Specific Accelerators

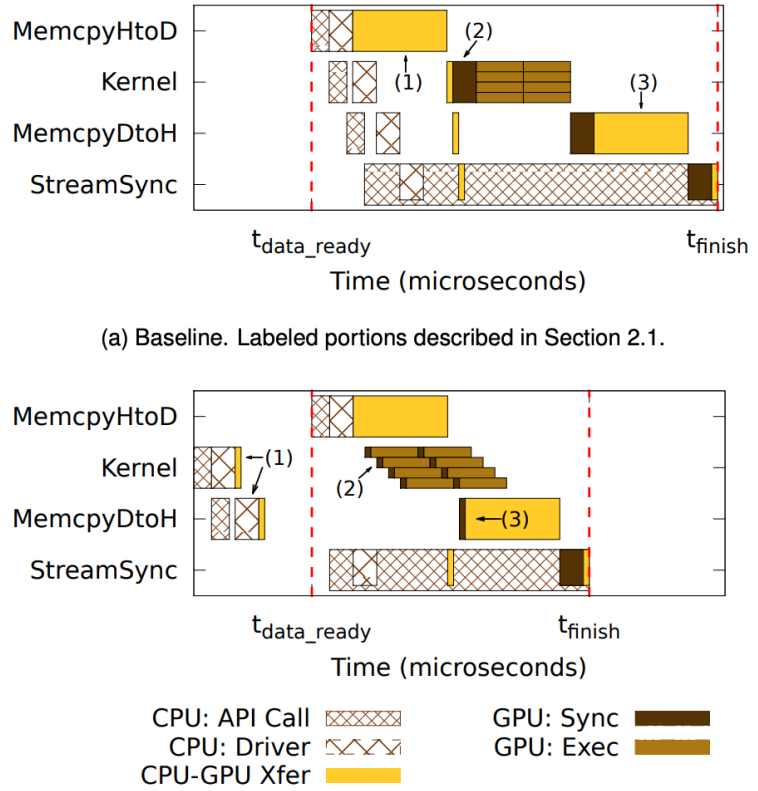
- Benefits:
  - Leverage static workload knowledge
  - Efficient hardware implementations
- Challenges:
  - Synchronization in a decoupled system





1) Full/Empty Bits Synchronization

- Use full-empty bits to track when regions of data have been transferred.
  - A full-empty bit per word
  - Producer writes only if the full-empty bit is empty, and sets it to full
  - Consumer reads only if the full-empty bit is set to full
- Pro:
  - Fine-grained synchronization
- Con:
  - Hardware cost



(a) Baseline. Labeled portions described in Section 2.1.

(b) Performance Improvement using the "full-overlap" scenario. Labeled portions described in Section 2.3.

Heterogeneous Element Processor, Smith'1982
Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization, HPCA'2013

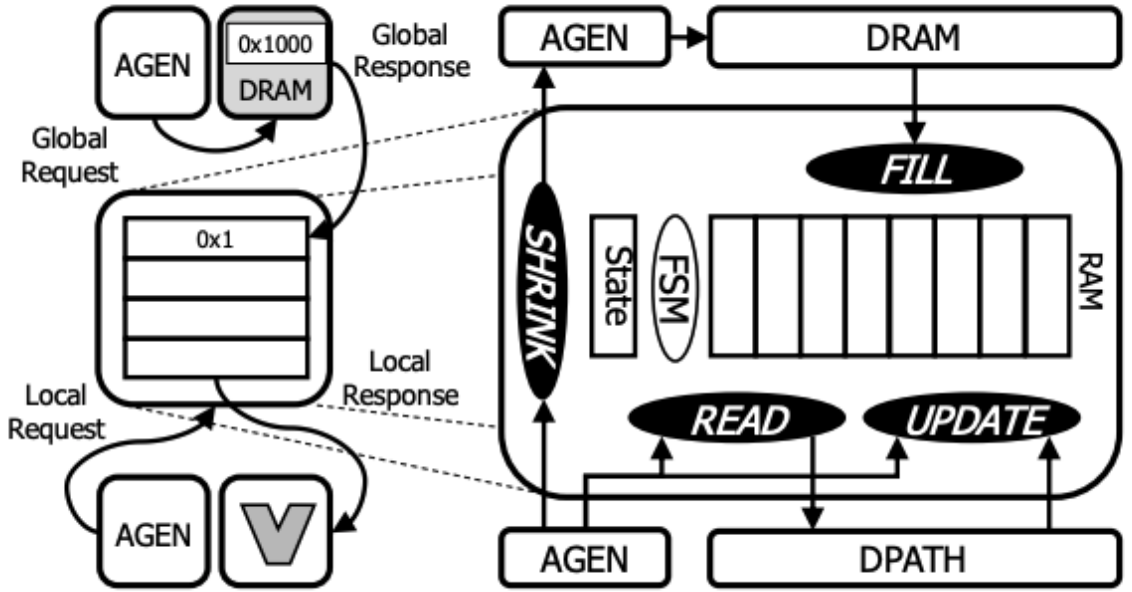
2) FIFO-based Synchronization

- FIFO is also EDDO.
  - Explicitly pushes/pops data
  - Decoupled requestor and consumer

- Features:
  - No need to have address accompanying the data
  - In-order fill
  - In-order read
  - Cannot do in-place update

3) Buffet-based Synchronization

- EDDO.
- Allows fine-grained overlap:
  - Fill-read overlap
- FIFO-like head-tail mechanisms together with:
  - Random read
  - In-place update
- Lifetime of staged data:

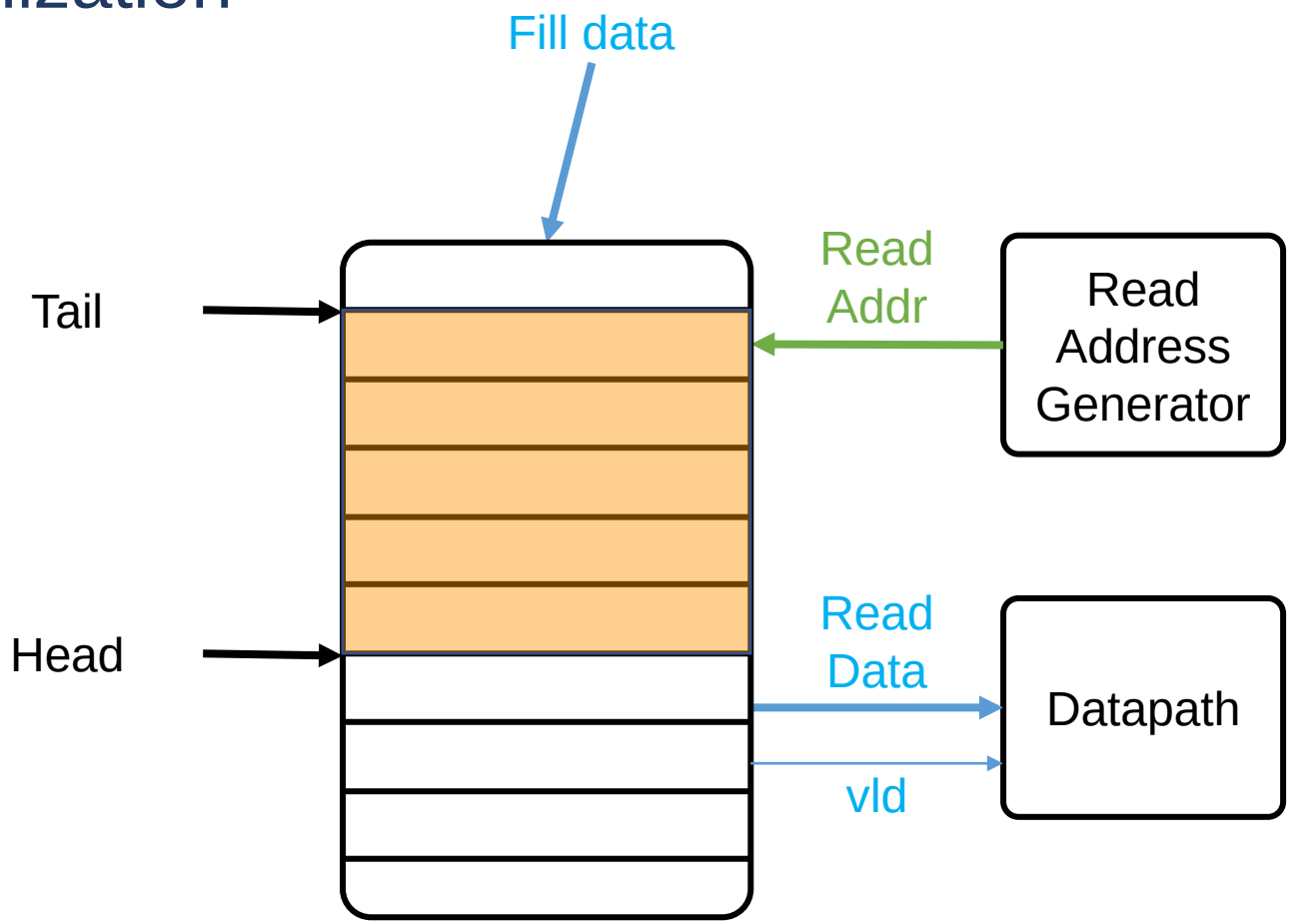


$$Fill \rightarrow (Read \rightarrow Update?)^* \rightarrow Read \rightarrow Shrink$$

Buffets, ASPLOS'2019

3) Buffet-based Synchronization

- Fill-read synchronization

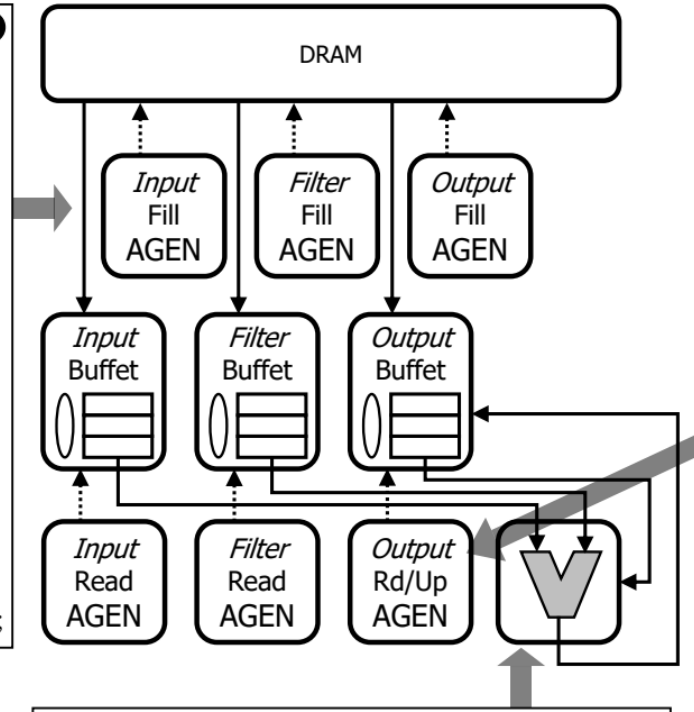


Example of A Buffet-based Accelerator

```
function FillInput()                                              ❶
  halo_size = O_TileSize - F_TileSize + 1;
  for nf : [0..F_NumTiles)
    tile_base = nf * F_TileSize;
    TransferInputTile(tile_base, halo_size);
    for no : [0..O_NumTiles)
      TransferInputTile(tile_base + halo_size +
                        no * O_TileSize, O_TileSize);

function FillFilter()
  for nf : [0..F_NumTiles)
    TransferFilterTile(nf * F_TileSize, F_TileSize);

function FillOutput()
  for nf : [0..F_NumTiles)
    for no : [0..O_NumTiles)
      TransferOutputTile(no * O_TileSize, O_TileSize);
```

All tile transfer functions above follow this form:

```
function TransferOutputTile(base, size)                           ❷
  wait _until(output_credit >= size);
  // This can be implemented as bulk transfer:
  for x : [0..size)
    output_buffet.Fill(output[base+x]);
  output_credit -= size;
```



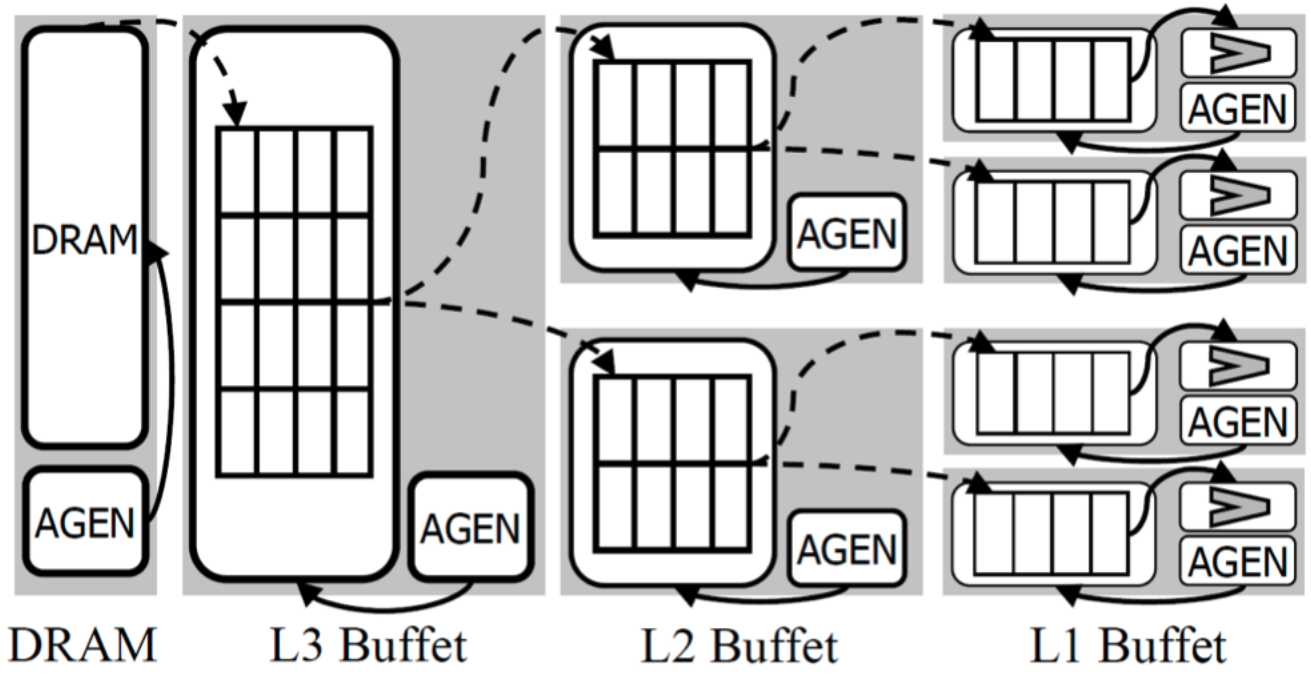

```
function Datapath()                                               ❹
  inp = input_buffet.read_rsp_out.Recv();
  wt = filter_buffet.read_rsp_out.Recv();
  psum = output_buffet.read_rsp_out.Recv();
  psum += inp * wt;
  output_buffet.update_data_in.Send(psum);
```

```
function ReadInput()                                              ❸
  for n : [0..F_NumTiles * O_NumTiles)
    for tf : [0..F_TileSize)
      for to : [0..O_TileSize)
        input_buffet.Read(tf+to);
    // Retain some overlap for next tile
    // (sliding window)
    input_buffet.Shrink(O_TileSize);

function ReadFilter()
  for n : [0..F_NumTiles * O_NumTiles)
    for tf : [0..F_TileSize)
      filter_buffet.Read(tf);
    filter_buffet.Shrink(F_TileSize);

function ReadAndUpdateOutput()
  for n : [0..F_NumTiles * O_NumTiles)
    for tf : [0..F_TileSize)
      for to : [0..O_TileSize)
        output_buffet.Read(to);
        output_buffet.update_idx_in.
            Send(to);
    // Drain of modified values omitted
    output_buffet.Shrink(O_TileSize);
```

Buffets, ASPLOS'2019

Example of A Buffet-based Accelerator

- Read response of Level *x* becomes the Fill input to Level *x*+1



Buffets, ASPLOS'2019

Review:

- Core computation in DNN
- Execution order of the core computation
- Hardware realization of the core computation
- Mapping DNNs to hardware
- This lecture: data transfer mechanisms across storage hierarchy
  - Guiding principles
  - Taxonomy:
    - Implicit vs Explicit
    - Coupled vs Decoupled
  - Case studies:
    - Cache, DAE, GPU shared memory, DMA