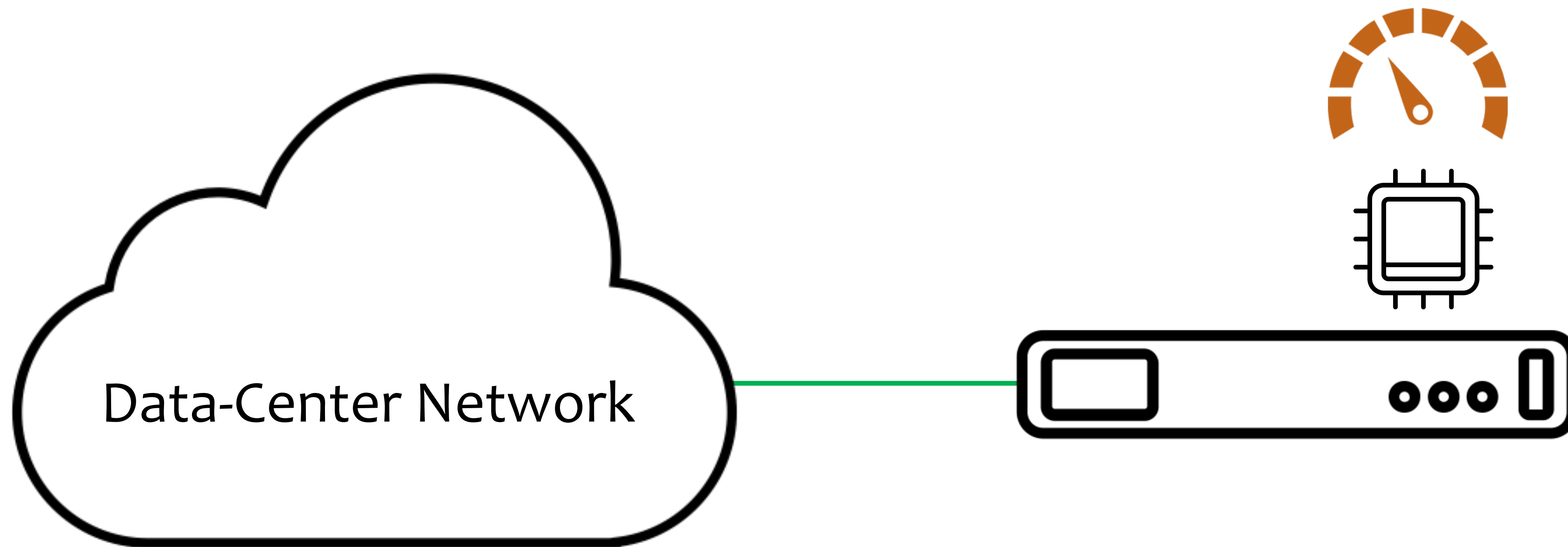
# Floem: Programming System for NIC-Accelerated Network Applications

**Phitchaya Mangpo Phothilimthana**
University of California, Berkeley

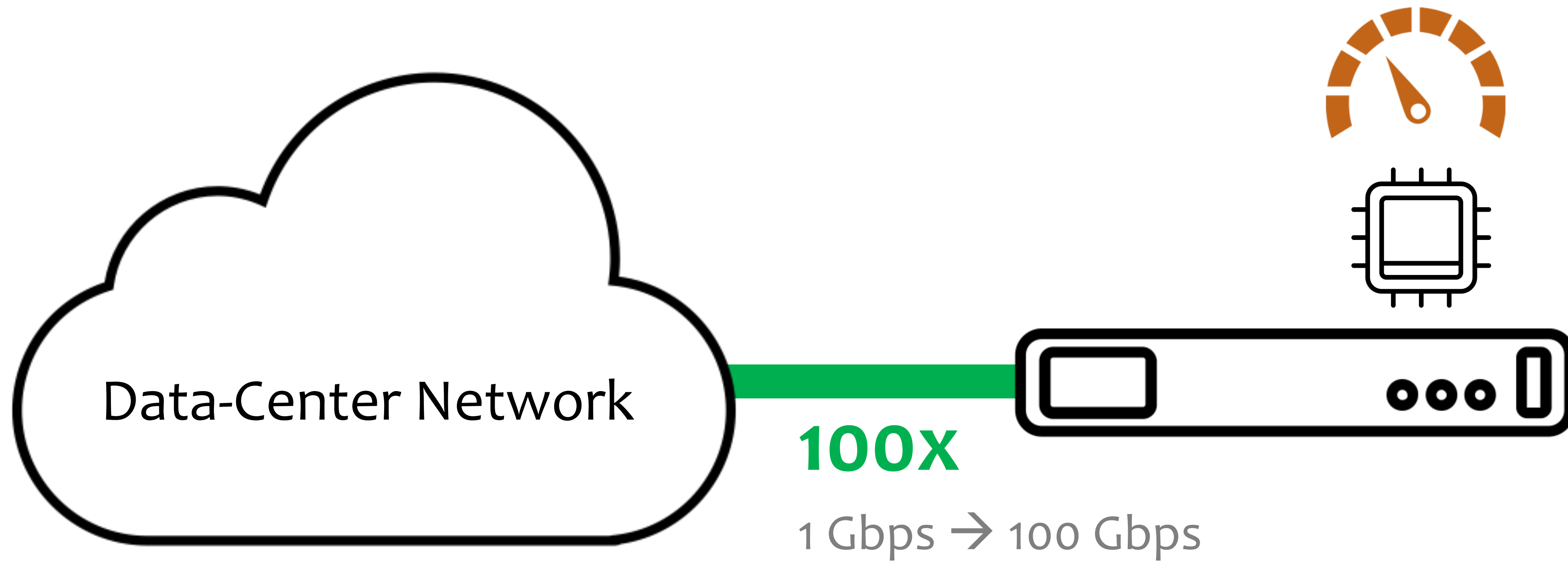
Ming Liu, Antoine Kaufmann, Ras Bodik, Tom Anderson
University of Washington

Simon Peter
UT Austin

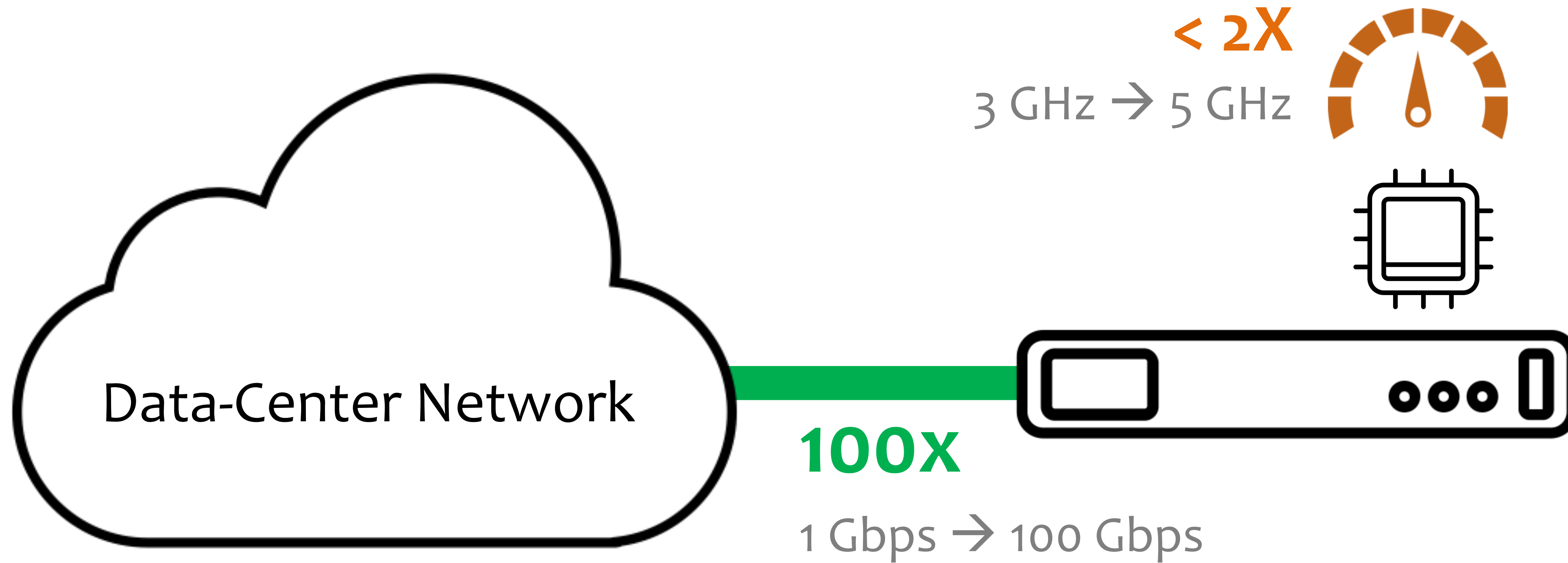
# Ethernet vs. CPU



# Ethernet vs. CPU



# Ethernet vs. CPU

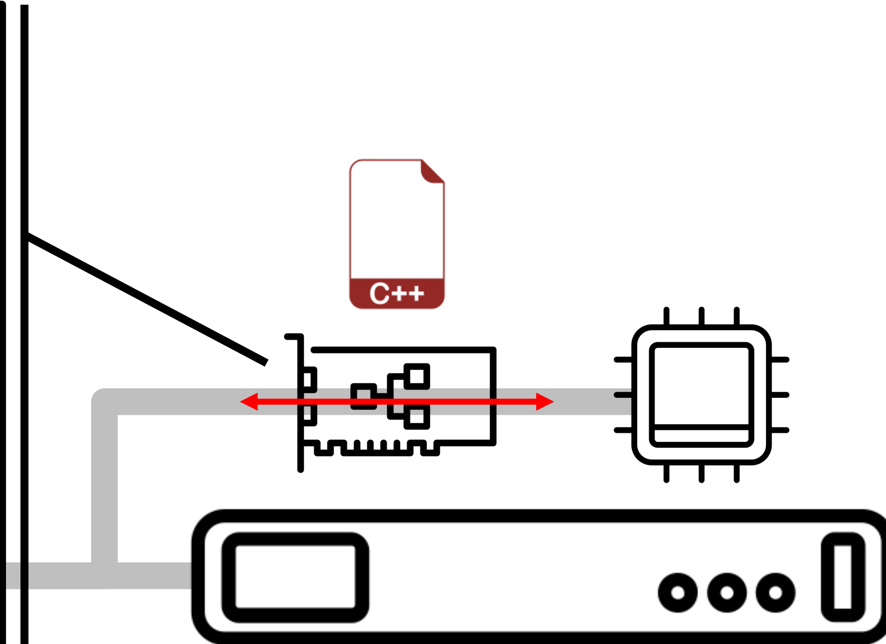


# Network Card (NIC)

**Wimpy multi-core processor**
- Cavium LiquidIO
- Netronome Agilio
- Mellanox BlueField

**Field-programmable gate array (FPGA)**
- Microsoft Catapult
- NetFPGA

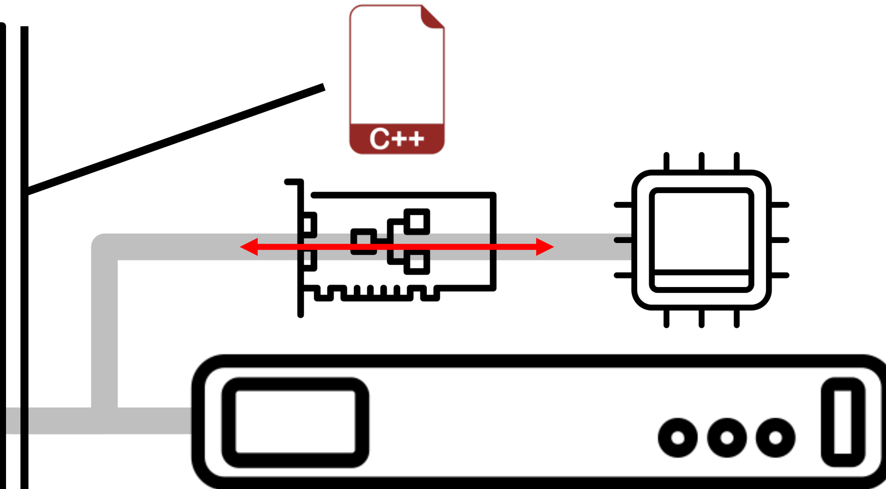
**Reconfigurable Match Table (RMT)**
- Bosshart et al. 2013
- Kaufmann et al. 2015



# NIC Offload

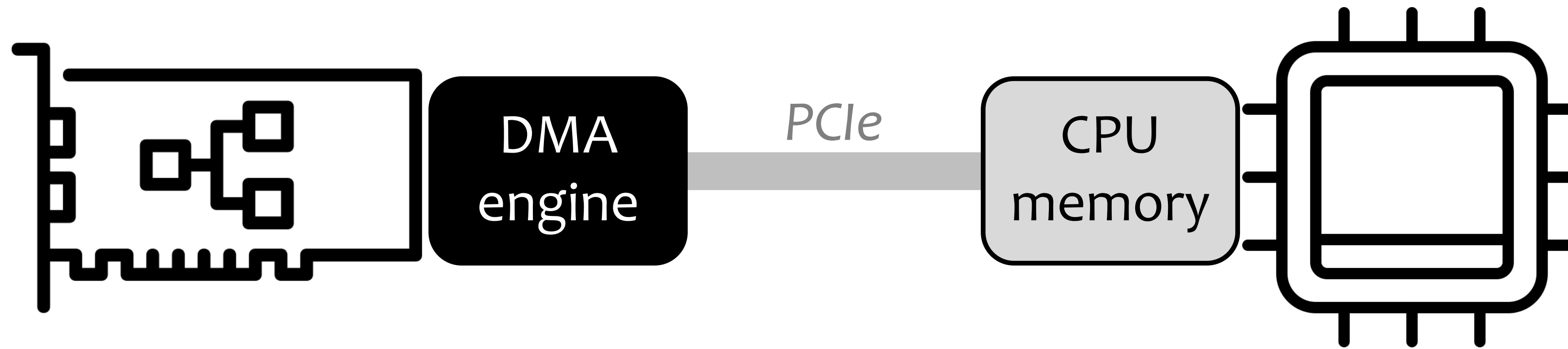
**Offload Computation**

- Fast path processing:
  filtering, classifying, caching, etc.
- Transformation:
  encryption/decryption, compression, etc.
- Steering
- Congestion control



## Offloading computation to a NIC requires a large amount of effort.

# Programming Platform



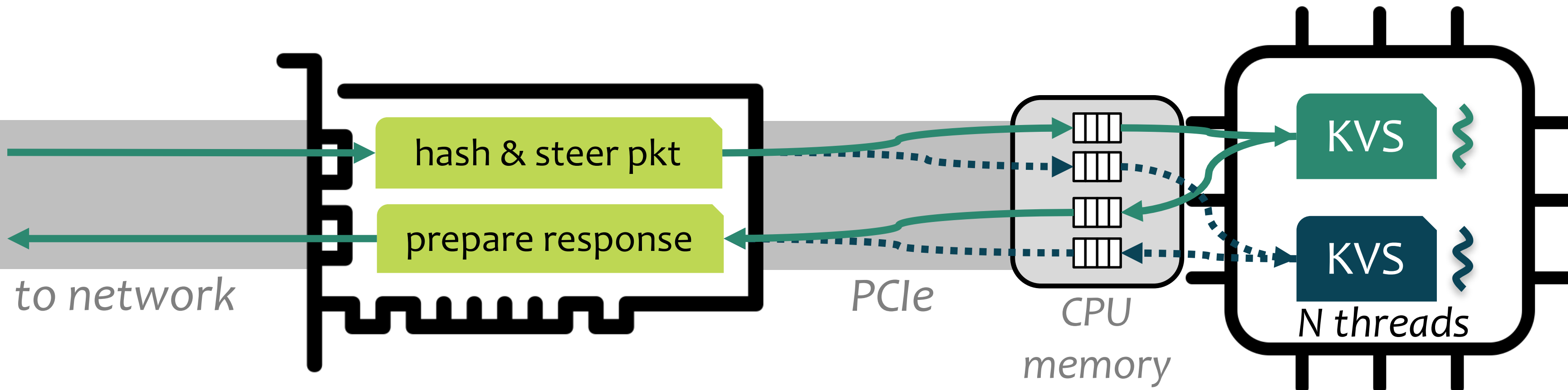

No cache coherence.
NIC can access CPU memory via **PCIe.**

**Cavium LiquidIO**
Slower cores
Lower power
No floating-point
Encryption co-processor
L1/L2 cache, DRAM, host memory

**Intel Xeon**
Faster cores
Higher power
Floating-point support
HW-accelerated instructions
L1/L2/L3 cache, DRAM, disk

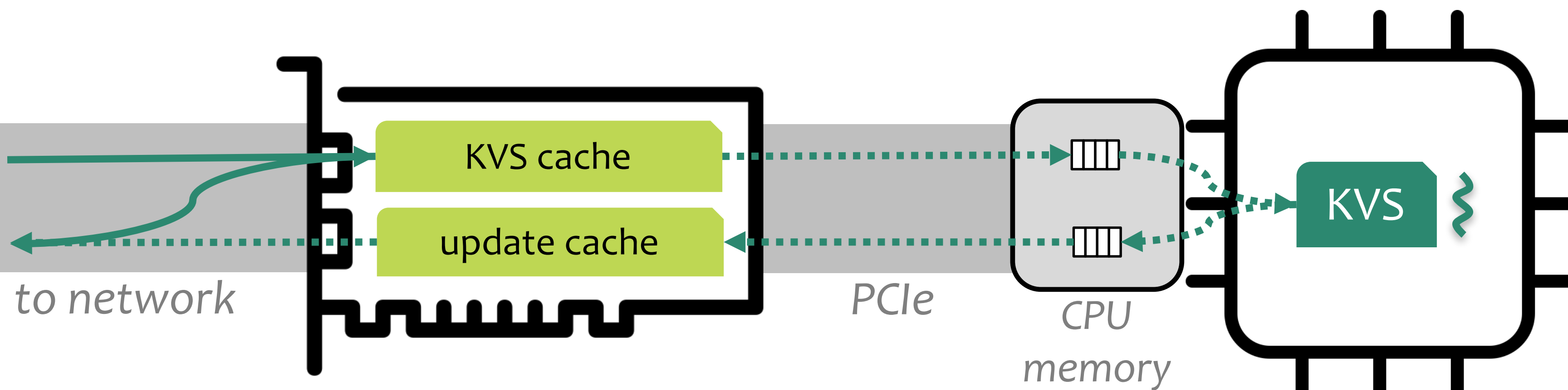
# Space of Offload Designs

**Example: Key-value store**




**Key-based steering**

- 30-45% higher throughput
- Require: multiple CPU cores
- *[Kaufmann et al. 2016]*



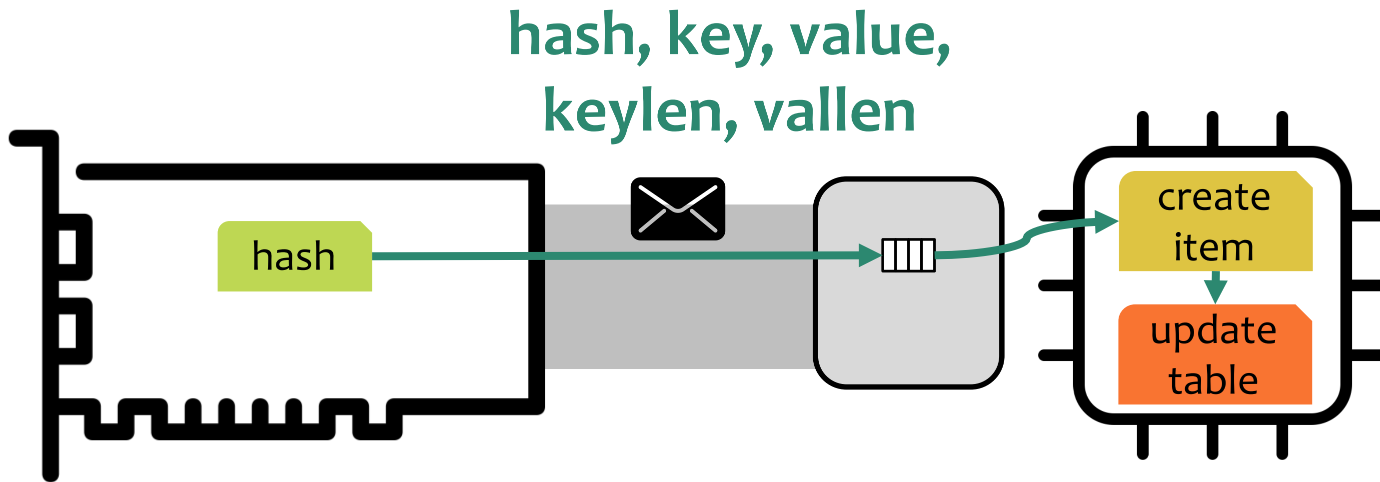

**Using NIC as Cache**

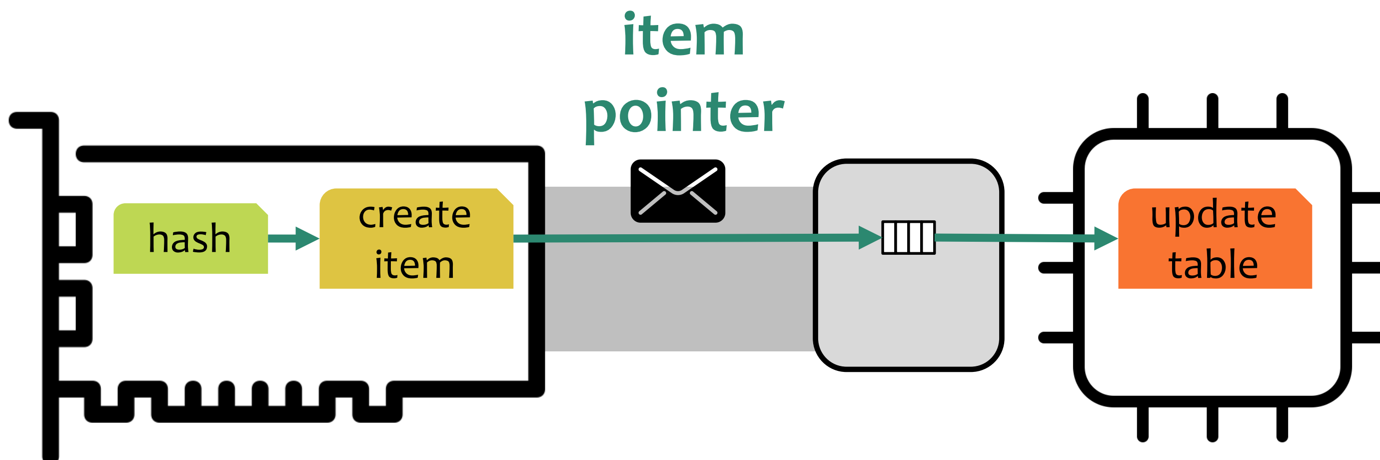
- 3x power efficiency
- Require: enough memory on NIC
- *[Li et al. 2017]*

**No one-size-fit-all offload.**
**Non-trivial to predict which offload is best.**

# Challenge: Packet Marshaling

**Example: Key-value store**




```
// Define what fields to send
struct set_request_entry {
    uint16_t flags;
    uint16_t len;
    uint32_t hash;
    uint32_t keylen;
    uint32_t vallen;
    uint8_t other[];
}
```

**tedious & error-prone**

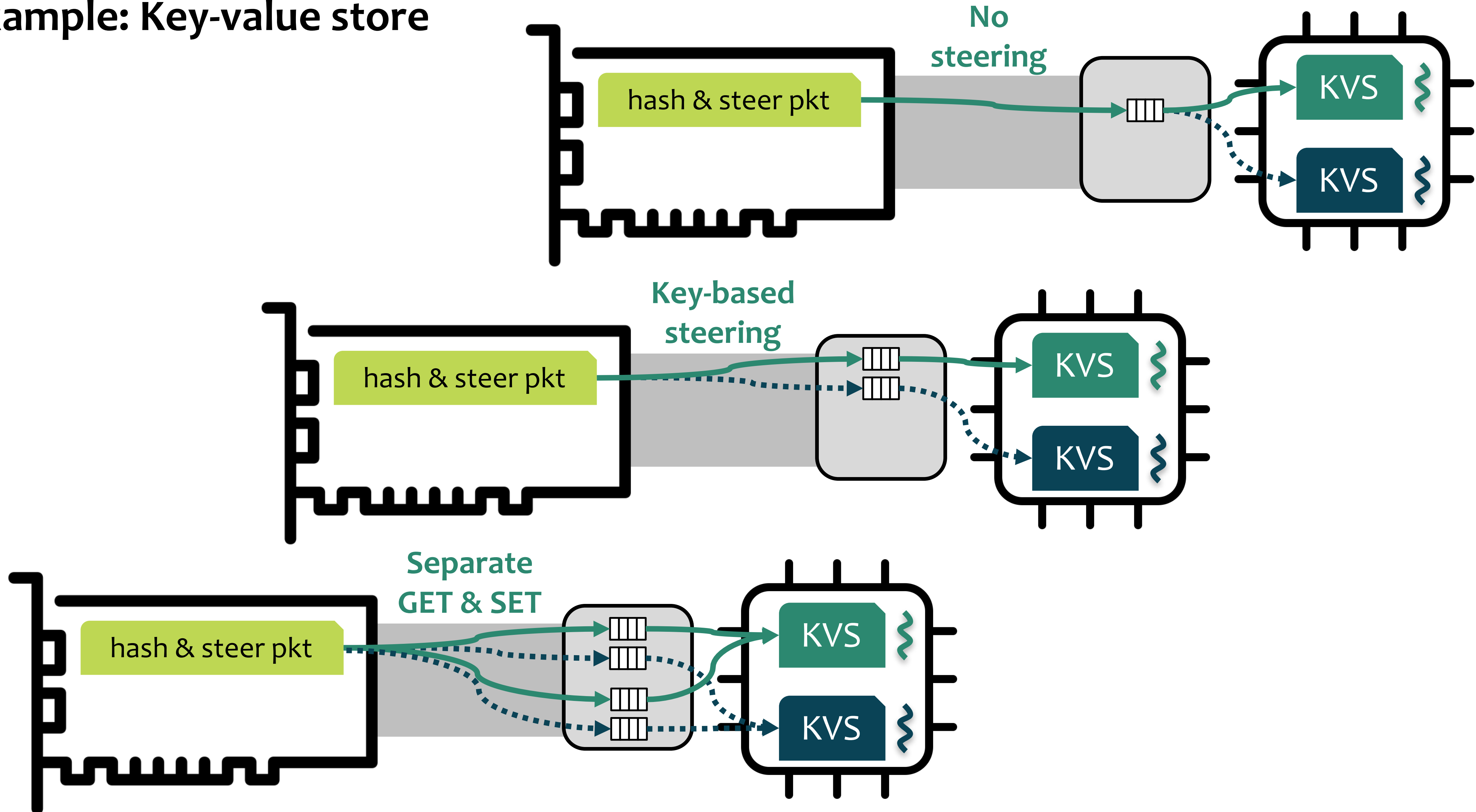
```
// Copy those fields
extra = it->keylen + it->vallen;
entry = queue_alloc(sizeof(*entry) + extra, SET);
entry->hash = hash;
entry->keylen = it->keylen;
entry->vallen = it->vallen;
memcpy(entry->other, it->key, extra);
```

```
struct set_request_entry {
    uint16_t flags;
    uint16_t len;
    uint64_t item;
}

entry = queue_alloc(sizeof(*entry), SET);
entry->item = ialloc_to_offset(item);
```

# Challenge: Communication Strategies

**Example: Key-value store**



**Exploring different offload designs requires a large amount of effort.**

# Floem

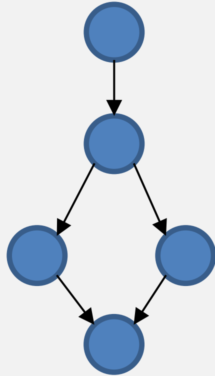
**DSL** makes it easy to explore alternative offloads.

**Compiler** minimizes communication and generates efficient code.

**Runtime** manages data transfer over PCIe.

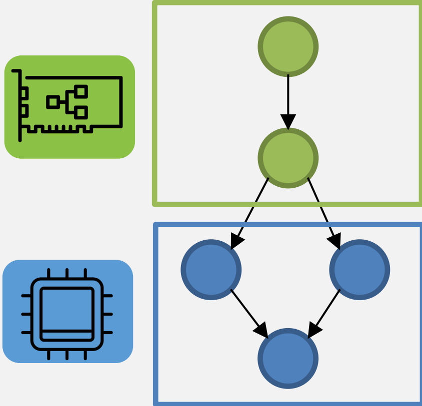
# Language Overview

**Data-flow programming model**



# Language Overview

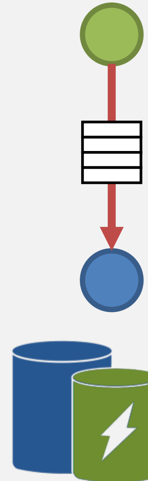
## Data-flow programming model



Extend to support:
- Heterogeneity
- Parallelism

## Contributions

***Goal: Explore offload designs***

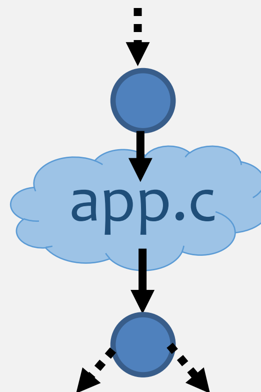


**1.** Inferred data transfer

**2.** Logical-to-physical queue mapping

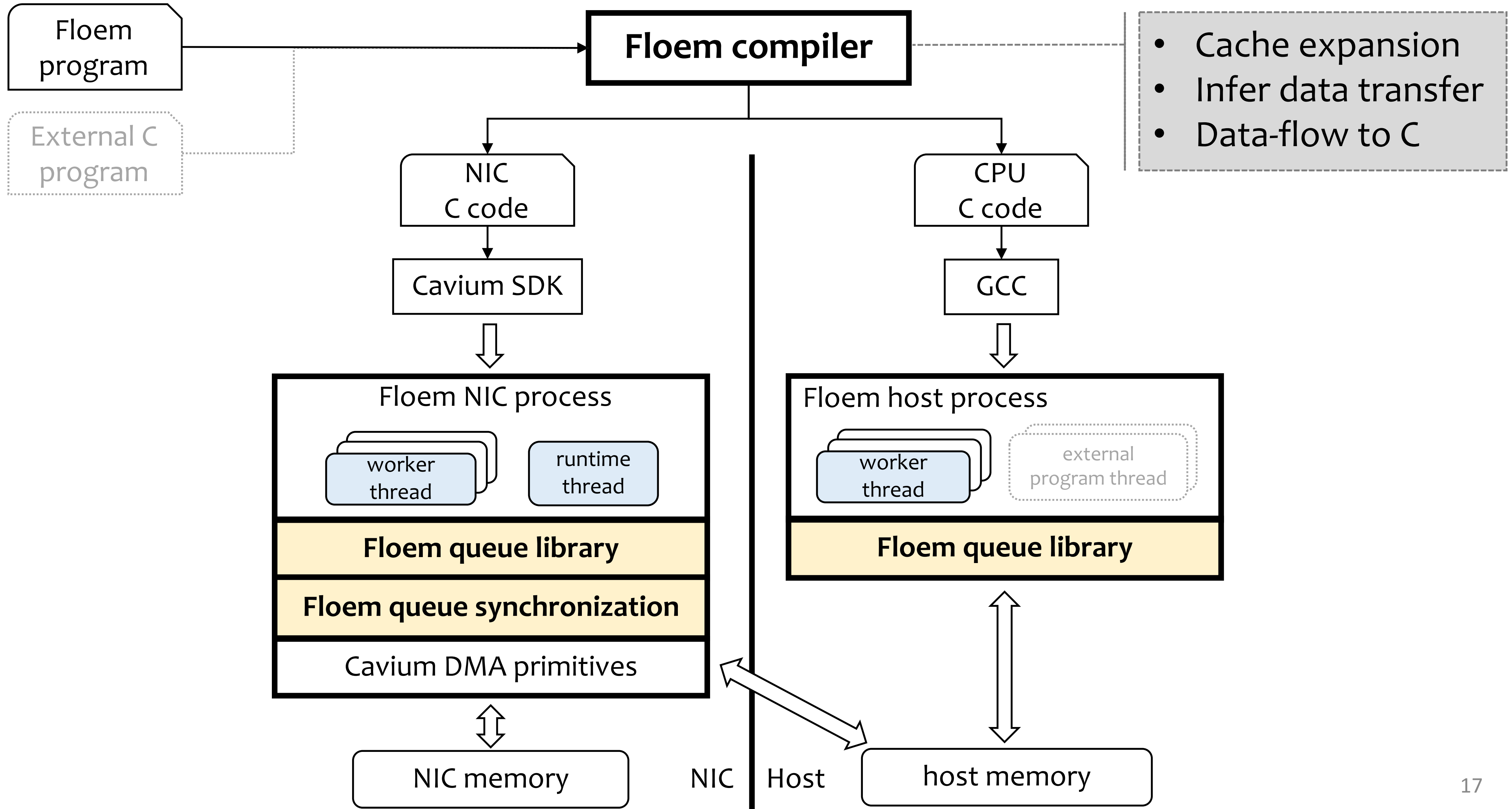
**3.** Caching construct

***Goal: Integration with existing app***



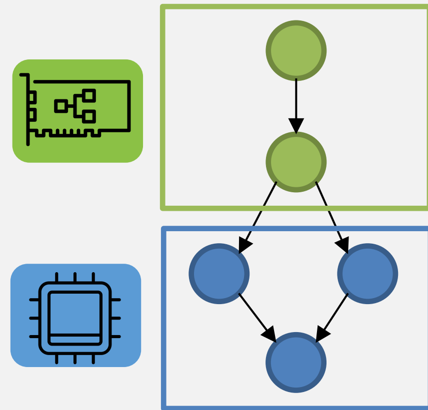

**4.** Interface to external programs

# Compiler & Runtime



# Data-Flow Model

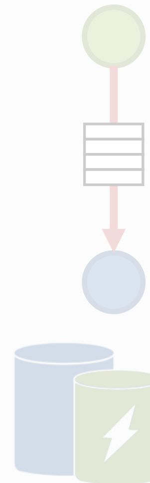
## Data-flow programming model



Extend to support:
- Heterogeneity
- Parallelism

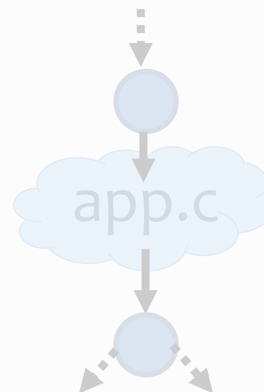
## Contributions

*Goal: Explore offload designs*



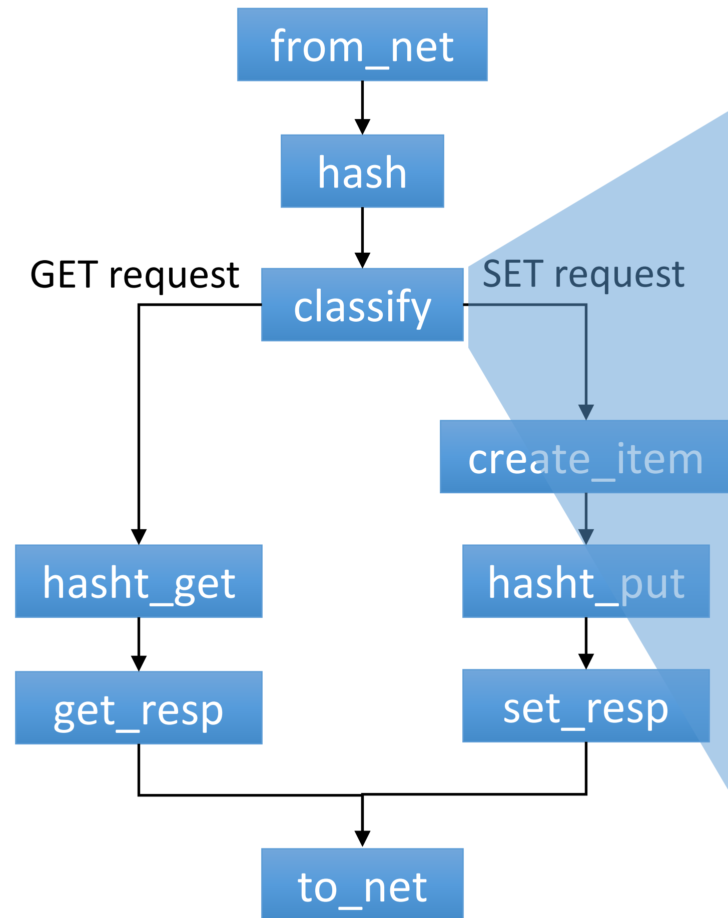
1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

*Goal: Integration with existing app*




4. Interface to external programs

# Data-Flow Model: Key-Value Store



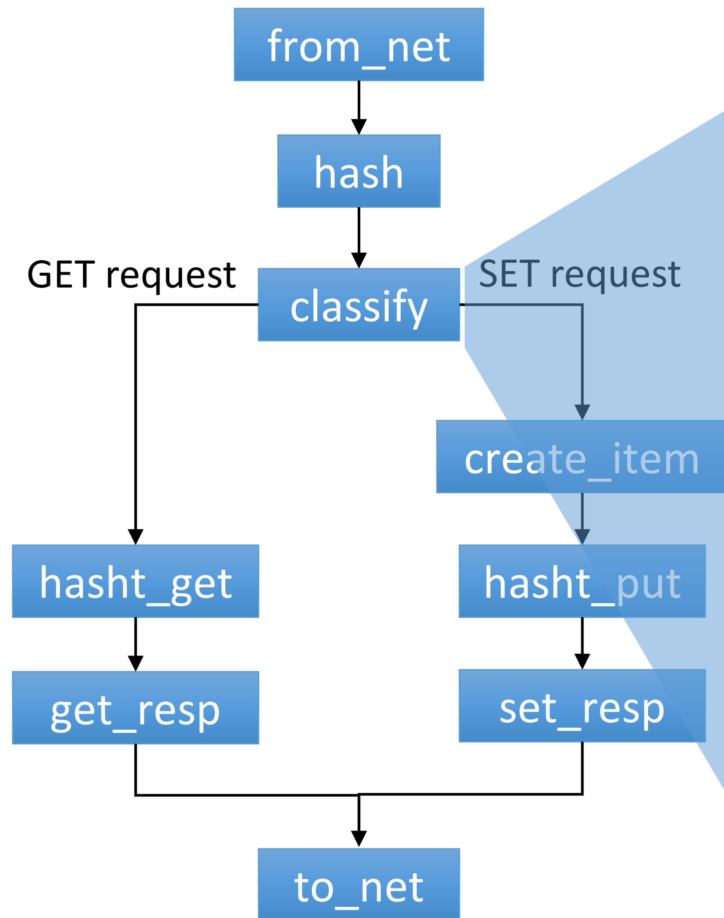

## Element

```
class Classify(Element):
  def configure(self):
    self.inp = Input(pointer(kvs_message))
    self.get = Output(pointer(kvs_message))
    self.set = Output(pointer(kvs_message))

  def impl(self):
    self.run_c(r'''
      // C code
      kvs_message *p = inp();
      uint8_t cmd = p->mcr.request.opcode;
      output switch {
        // switch --> emit one output port
        case (cmd == PROTOCOL_BINARY_CMD_GET): get(p);
        case (cmd == PROTOCOL_BINARY_CMD_SET): set(p);
      }''')

classify = Classify()  # Instantiate an element
```

# Data-Flow Model: Key-Value Store



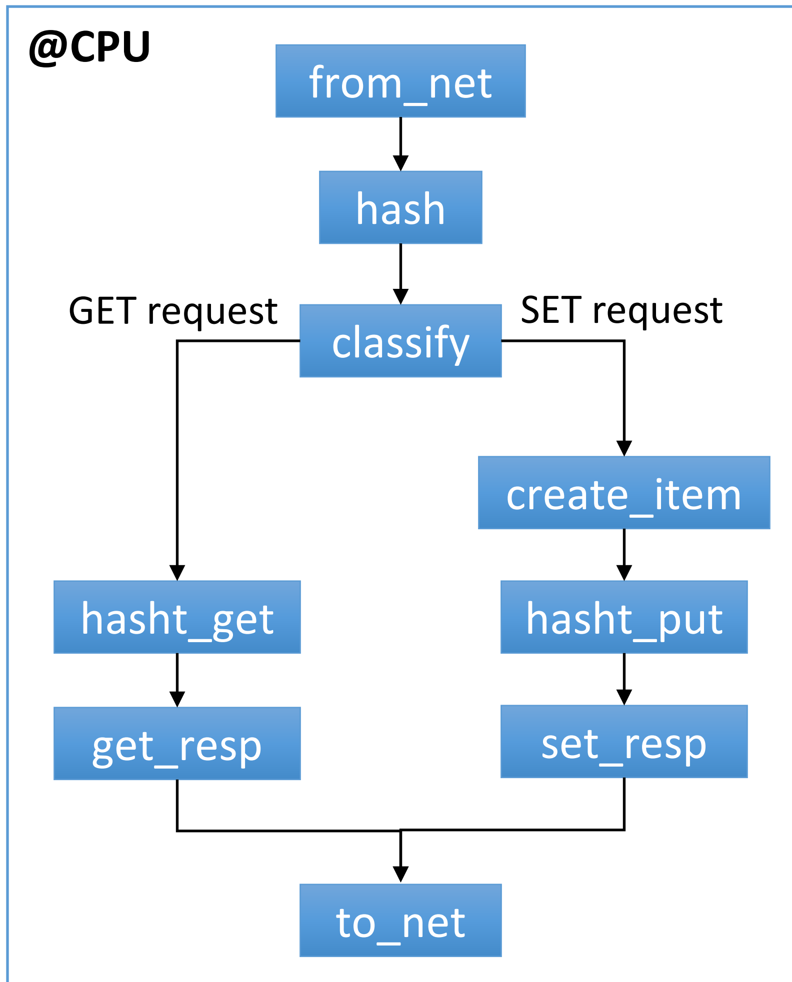

## Element

```
class Classify(Element):
  def configure(self):
    self.inp = Input(pointer(kvs_message))
    self.get = Output(pointer(kvs_message))
    self.set = Output(pointer(kvs_message))

  def impl(self):
    self.run_c(r'''
      // C code
      kvs_message *p = inp();
      uint8_t cmd = p->mcr.request.opcode;
      output switch {
        // switch --> emit one output port
        case (cmd == PROTOCOL_BINARY_CMD_GET): get(p);
        case (cmd == PROTOCOL_BINARY_CMD_SET): set(p);
      }''')

classify = Classify()  # Instantiate an element
```

# Data-Flow Model

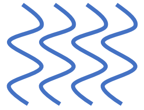


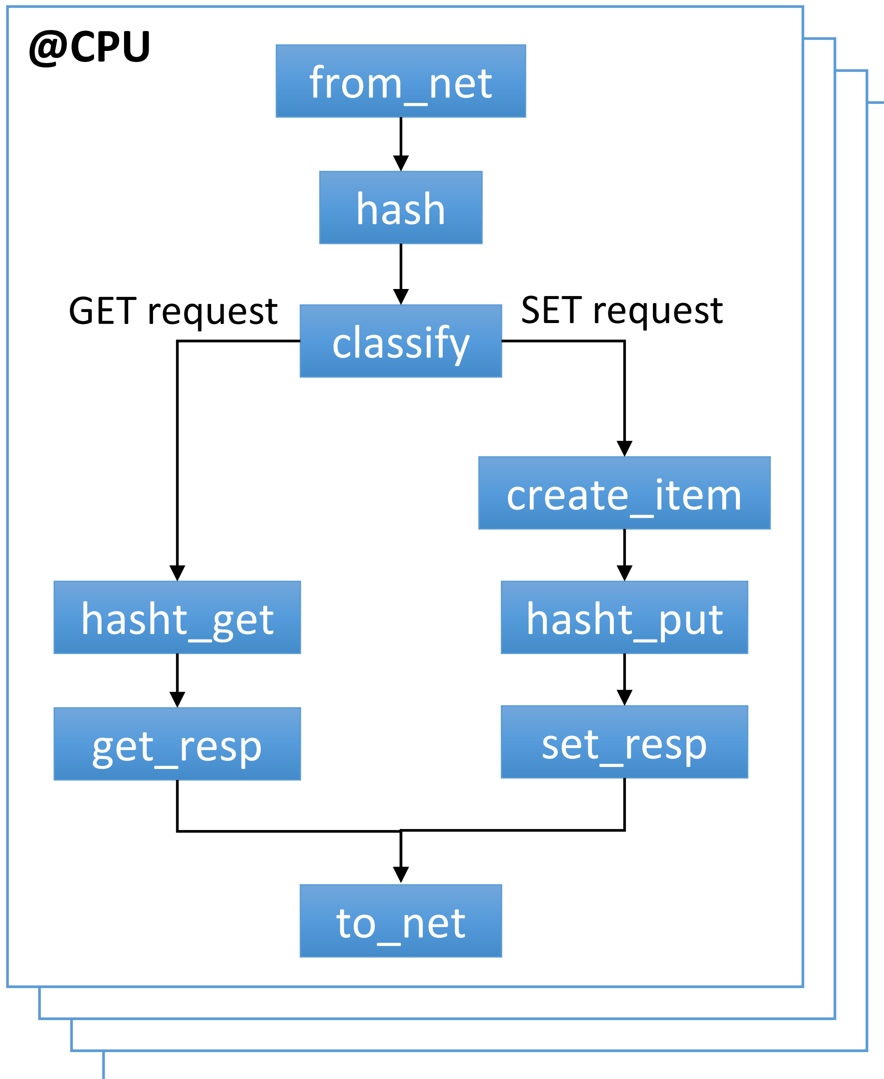

## Segment

```
class Seg1(Segment):
  def impl(self):
    from_net >> hash >> classify
    classify.get >> hasht_get >> get_resp >> to_net
    classify.set >> item >> hasht_put >> set_resp \\
      >> to_net

s1 = Seg1()
```

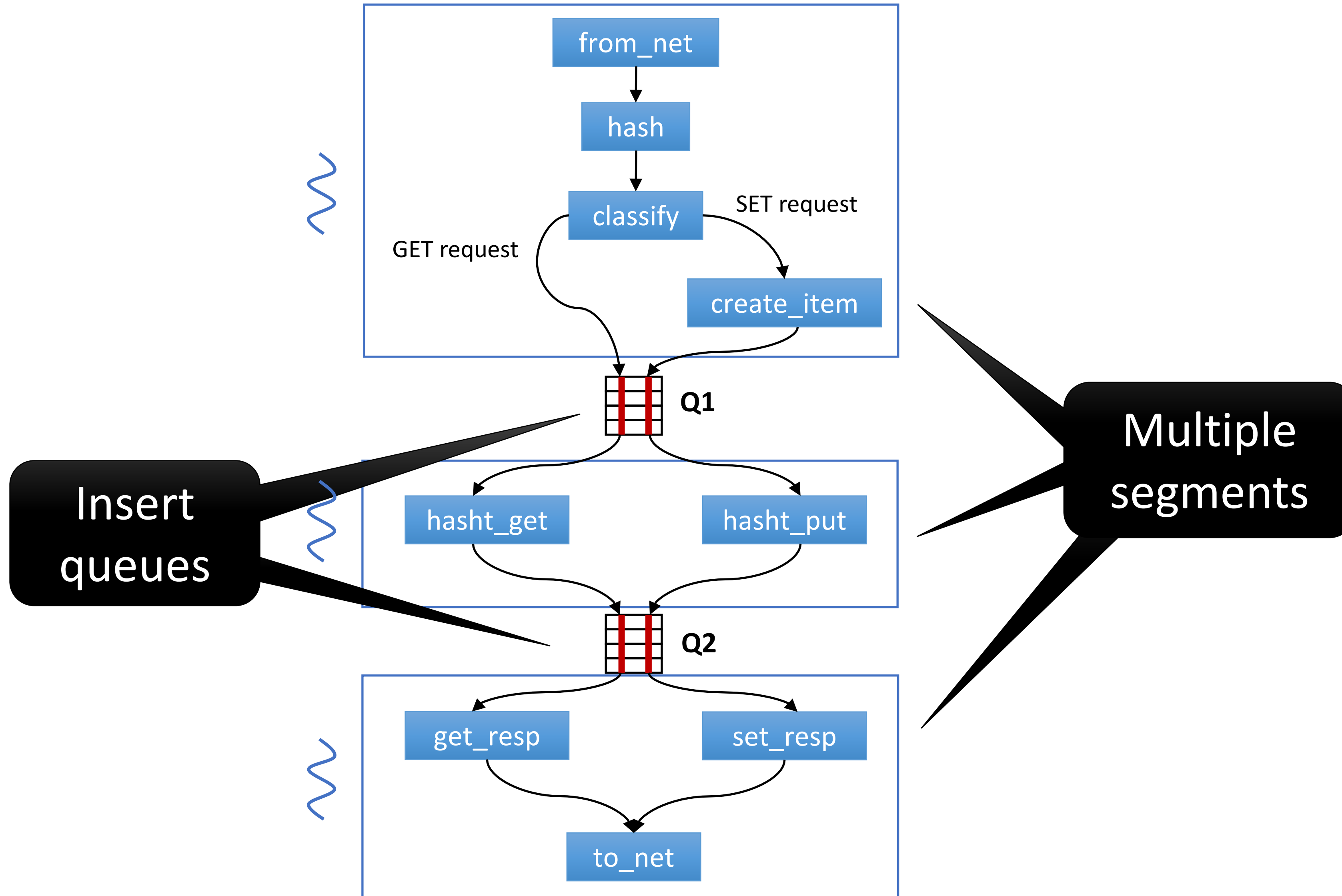
# Data Parallelism





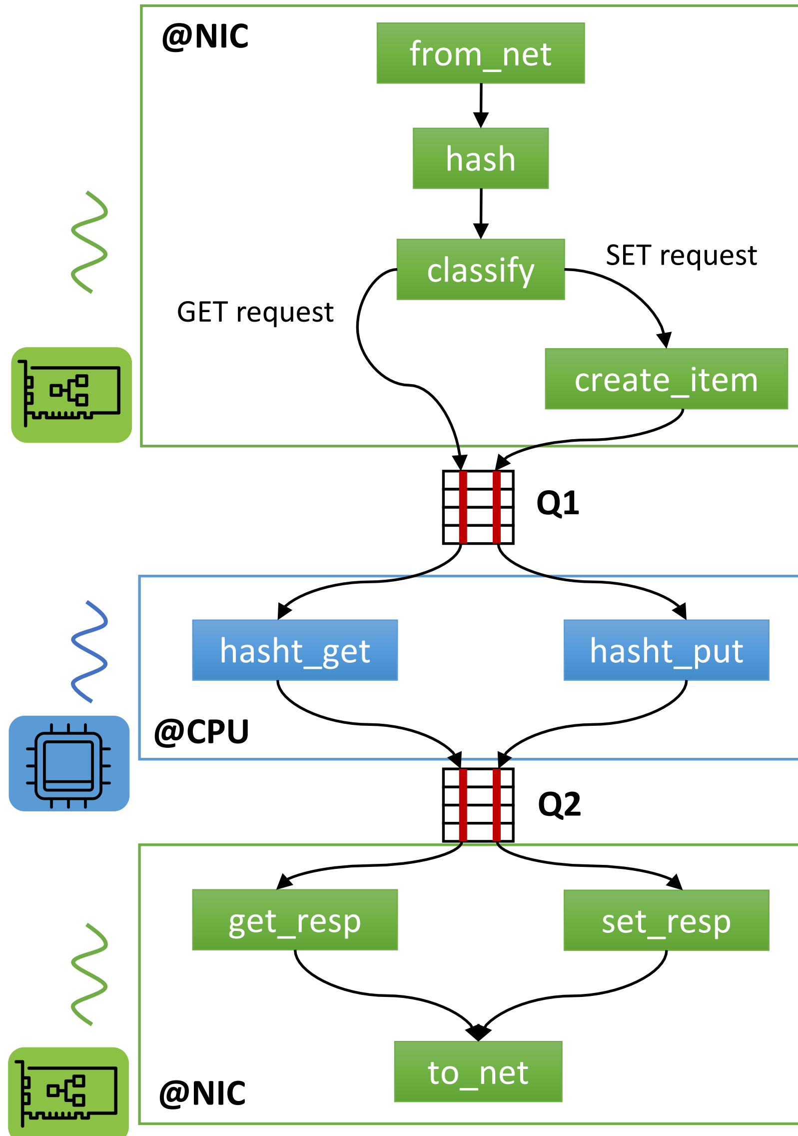

```
Seg1(cores=[0,1,2,3])
```

# Pipeline Parallelism




Multiple segments

# NIC Offload



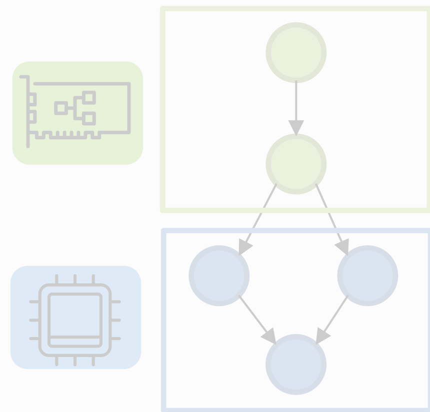

`Seg1(device=NIC)`

`Seg2(device=CPU)`

`Seg3(device=NIC)`

# Inferred Data Transfer

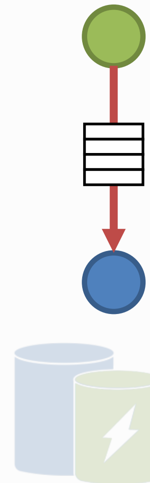
## Data-flow programming model



Extend to support:
- Heterogeneity
- Parallelism

## Contributions

*Goal: Explore offload designs*

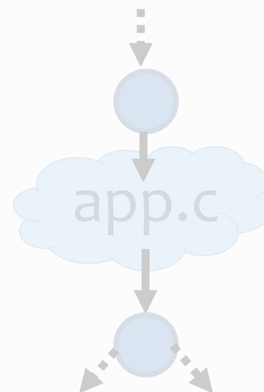


**1.** Inferred data transfer

**2.** Logical-to-physical queue mapping

**3.** Caching construct

*Goal: Integration with existing app*

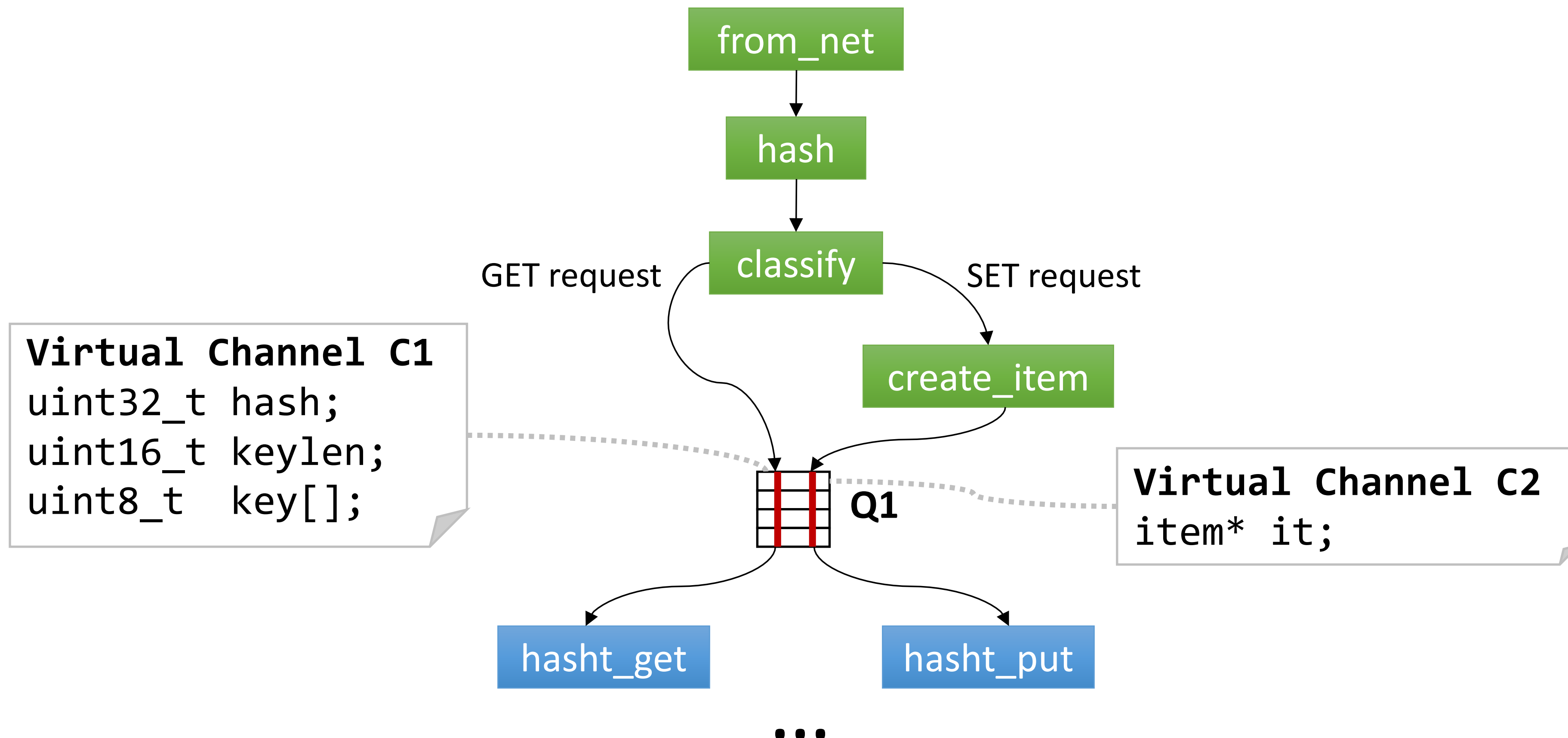



**4.** Interface to external programs

# Solution: Infer Fields to Send

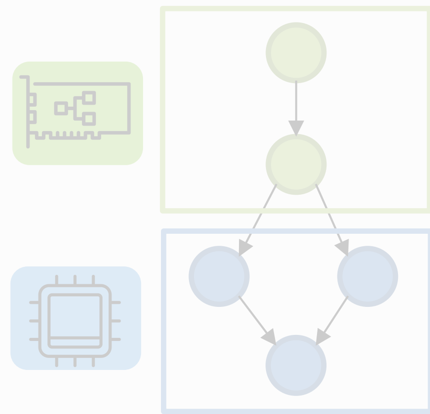
**Per-packet state:** a packet and its metadata can be accessed anywhere in the program.

**Compiler** infers which fields of packet and metadata to send.



# Logical-to-Physical Queue Mapping

## Data-flow programming model

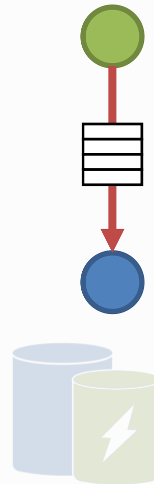


Extend to support:

- Heterogeneity
- Parallelism

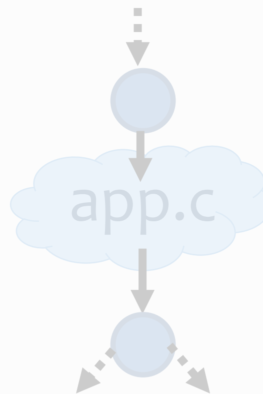
## Contributions

*Goal: Explore offload designs*



1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

*Goal: Integration with existing app*




4. Interface to external programs

# Queue Construct

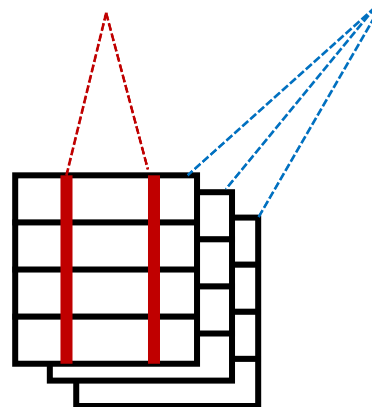
**Observation:** Different communication strategies can be expressed by mapping logical queues to physical queues.

- Degrees of resource sharing
- Dynamic packet steering
- Packet ordering

**Solution:** Queue construct with explicit logical-to-physical queue mapping.

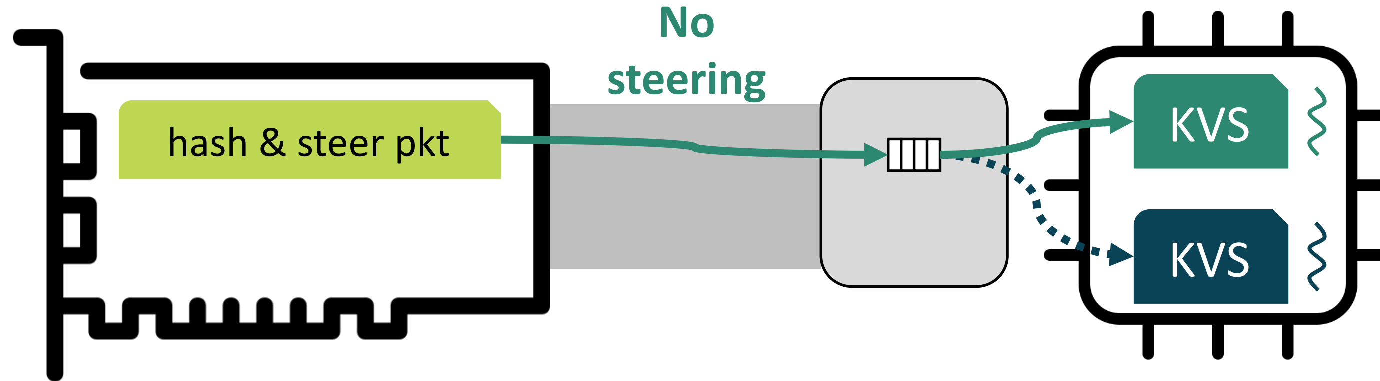
*logical queues* *physical queues*

```
Queue(channels=2,instances=3)
```



# No Steering

Strategy:




Program:

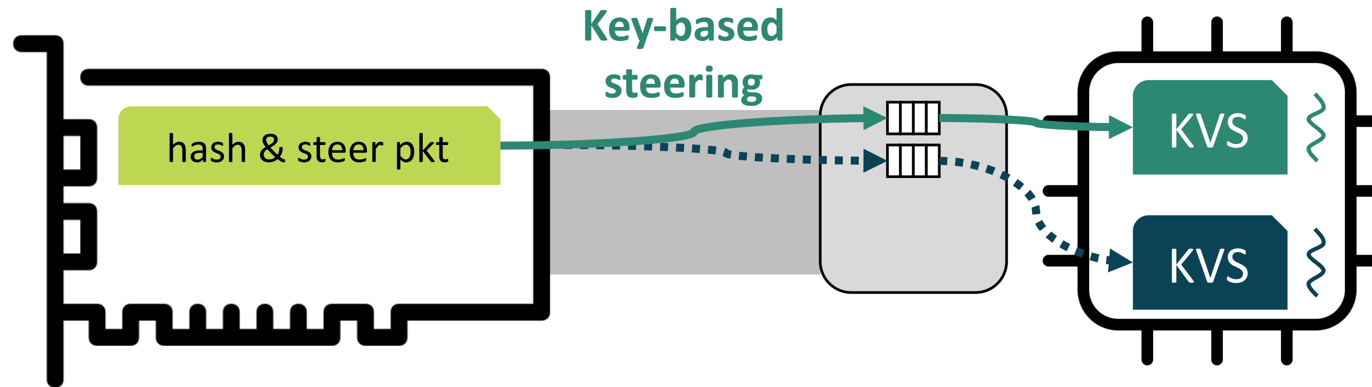


```
class Seg2(Segment):
  def impl(self):

    Q1.deq[0] >> hasht_get >> ...
    Q1.deq[1] >> hasht_put >> ...
```

# Key-Based Steering

Strategy:




Program:

...

classify

GET request

SET request

create_item

Q1

hasht_get

hasht_put

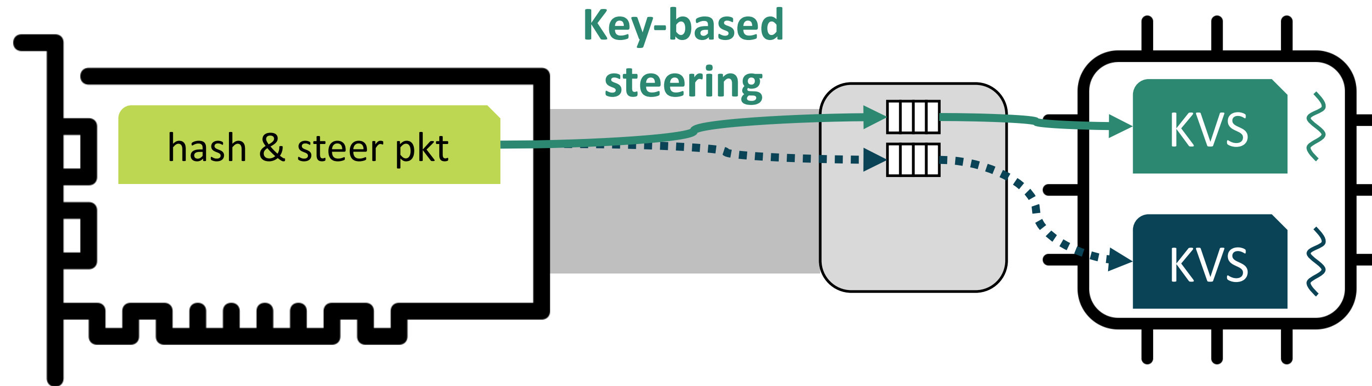
...

```
class Seg2(Segment):
  def impl(self):

    Q1.deq[0] >> hasht_get >> ...
    Q1.deq[1] >> hasht_put >> ...
```

# Key-Based Steering

Strategy:




Program:

queue_id
classify
GET request
SET request
create_item
Q1
hasht_get
hasht_put
...

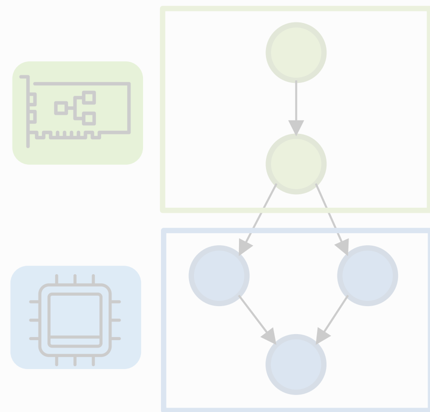
**Key-based steering:**

```
state.qid = state.hv % 2
```

```
class Seg2(Segment):
  def impl(self):
    self.core_id >> Q1.qid
    Q1.deq[0] >> hasht_get >> ...
    Q1.deq[1] >> hasht_put >> ...
```

# Caching Construct

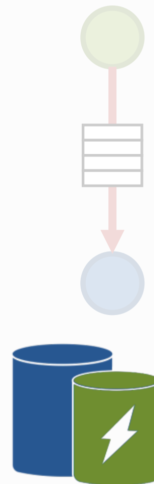
## Data-flow programming model



Extend to support:
- Heterogeneity
- Parallelism

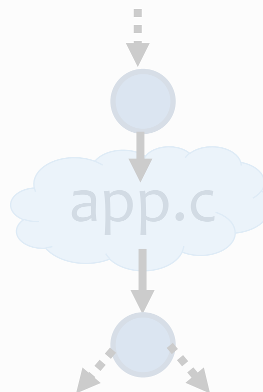
## Contributions

*Goal: Explore offload designs*



1. Inferred data transfer
2. Logical-to-physical queue mapping
3. Caching construct

*Goal: Integration with existing app*

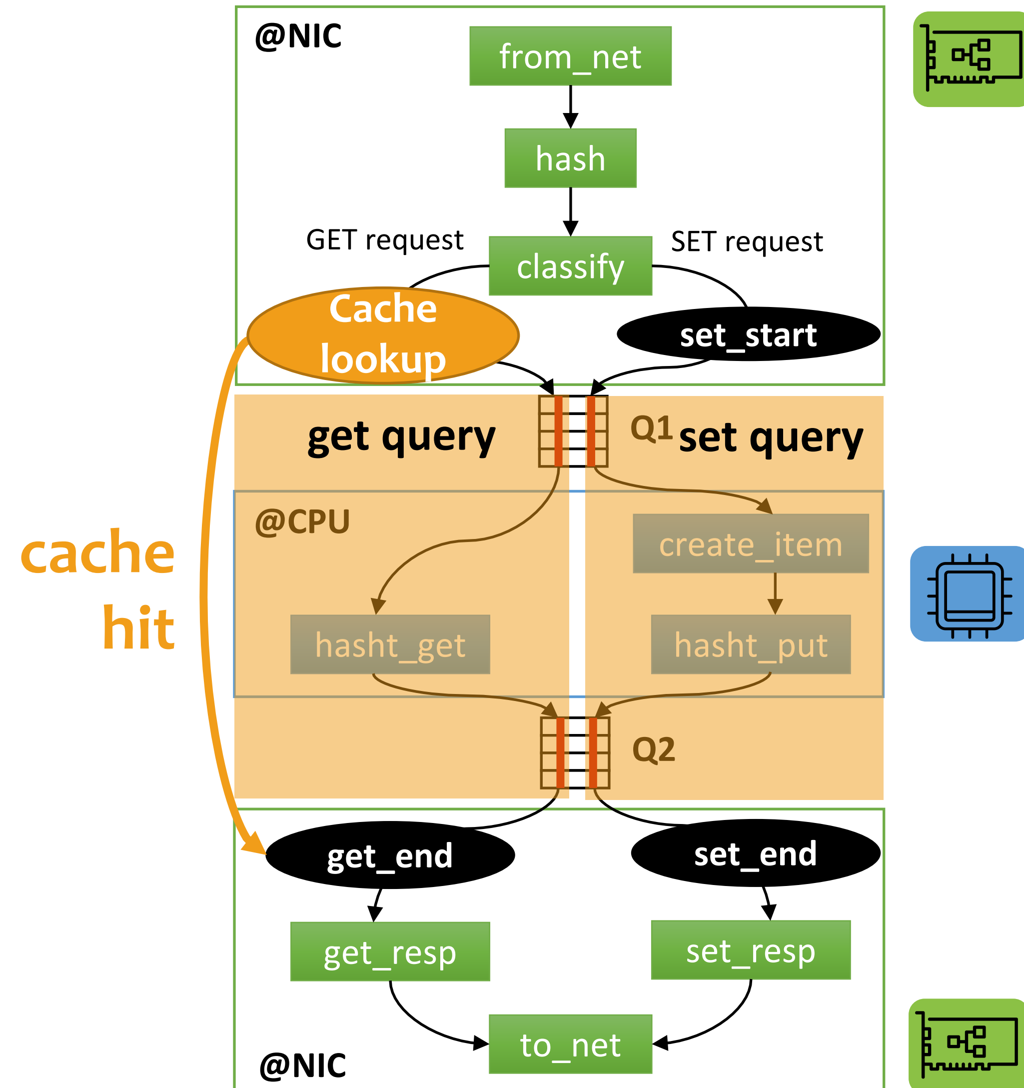



4. Interface to external programs

# Caching Construct

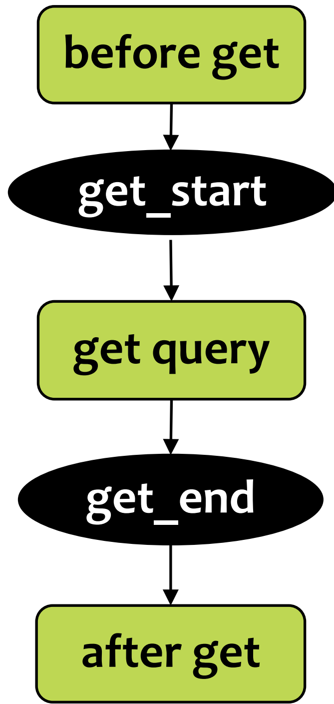
Difficult to implement a complete cache protocol:

- Maintain consistency of data on NIC and CPU
- High performance

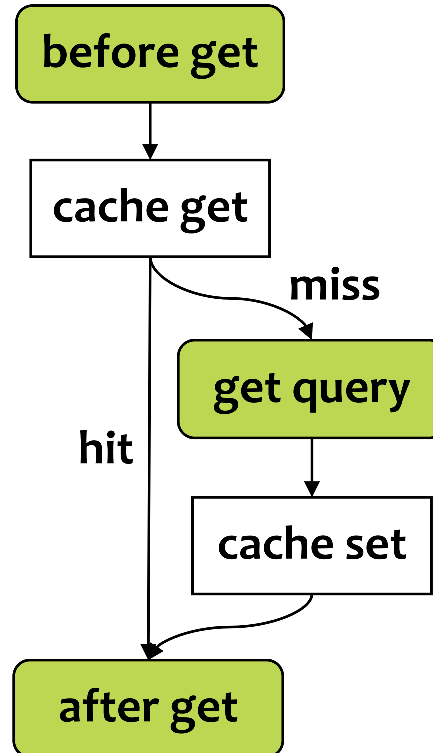


# Get Expansion

User's program

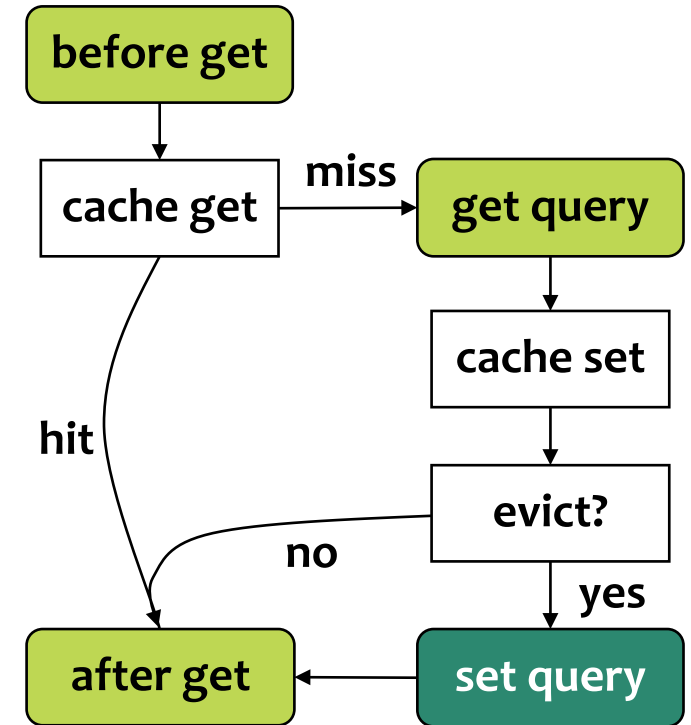



Write-through



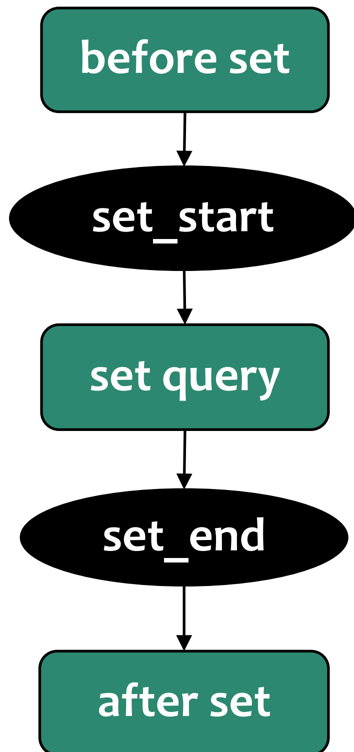

OR

Write-back

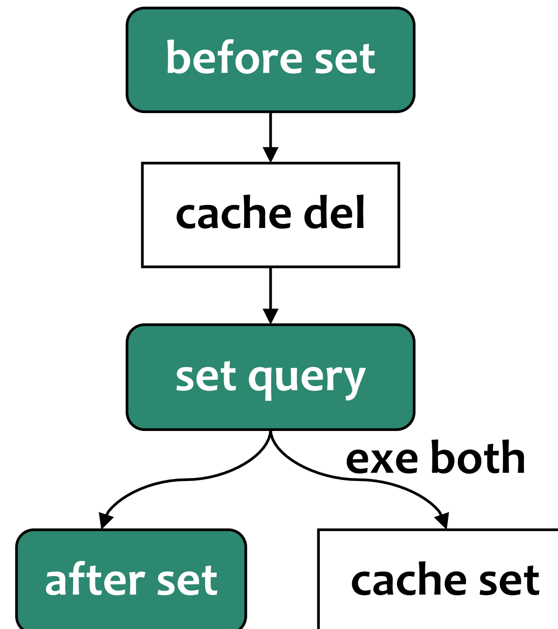


# Set Expansion

User's program

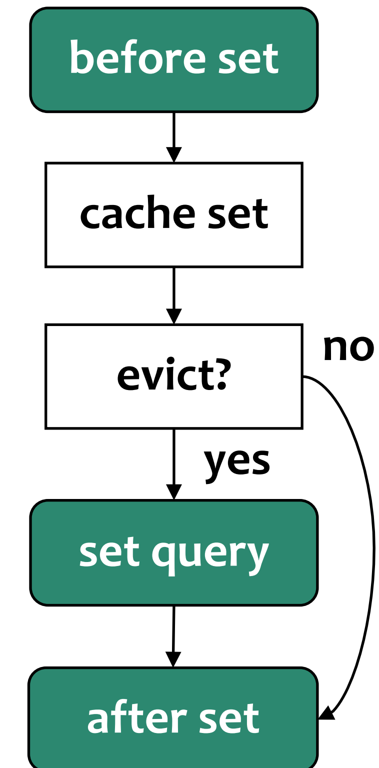



Write-through




OR

Write-back

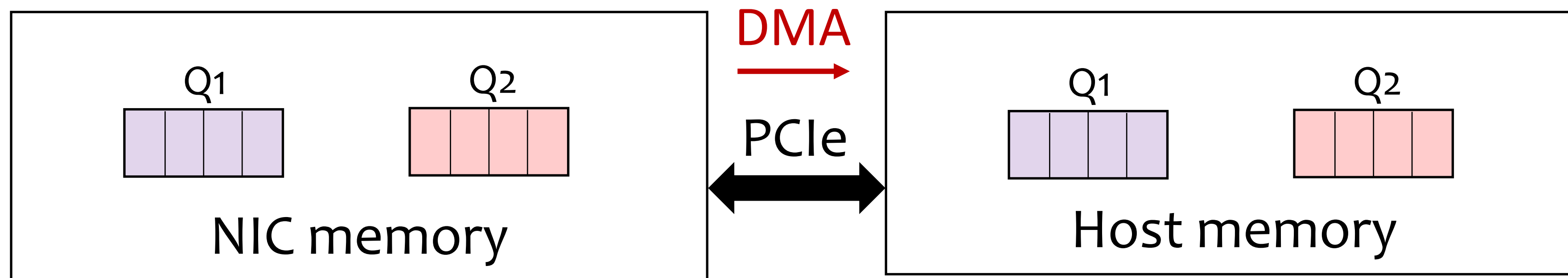


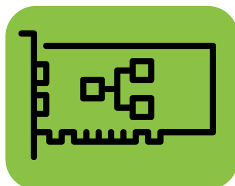
# Runtime & Communication

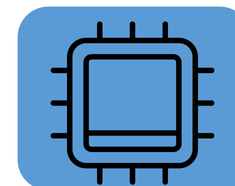
# Queue Implementation Challenge

For performance, require:

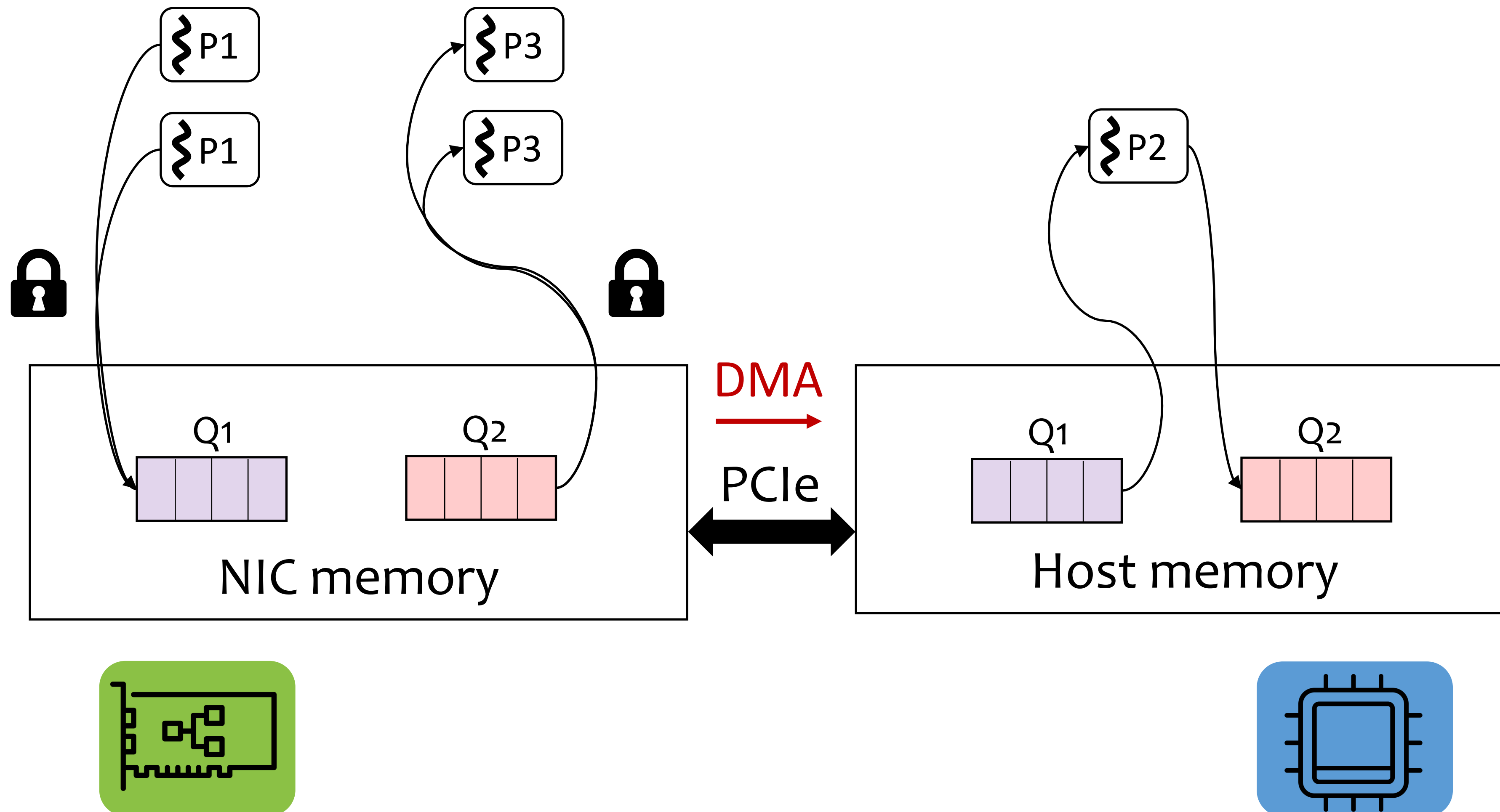
- I/O batching
- overlapping DMA operations with useful computation



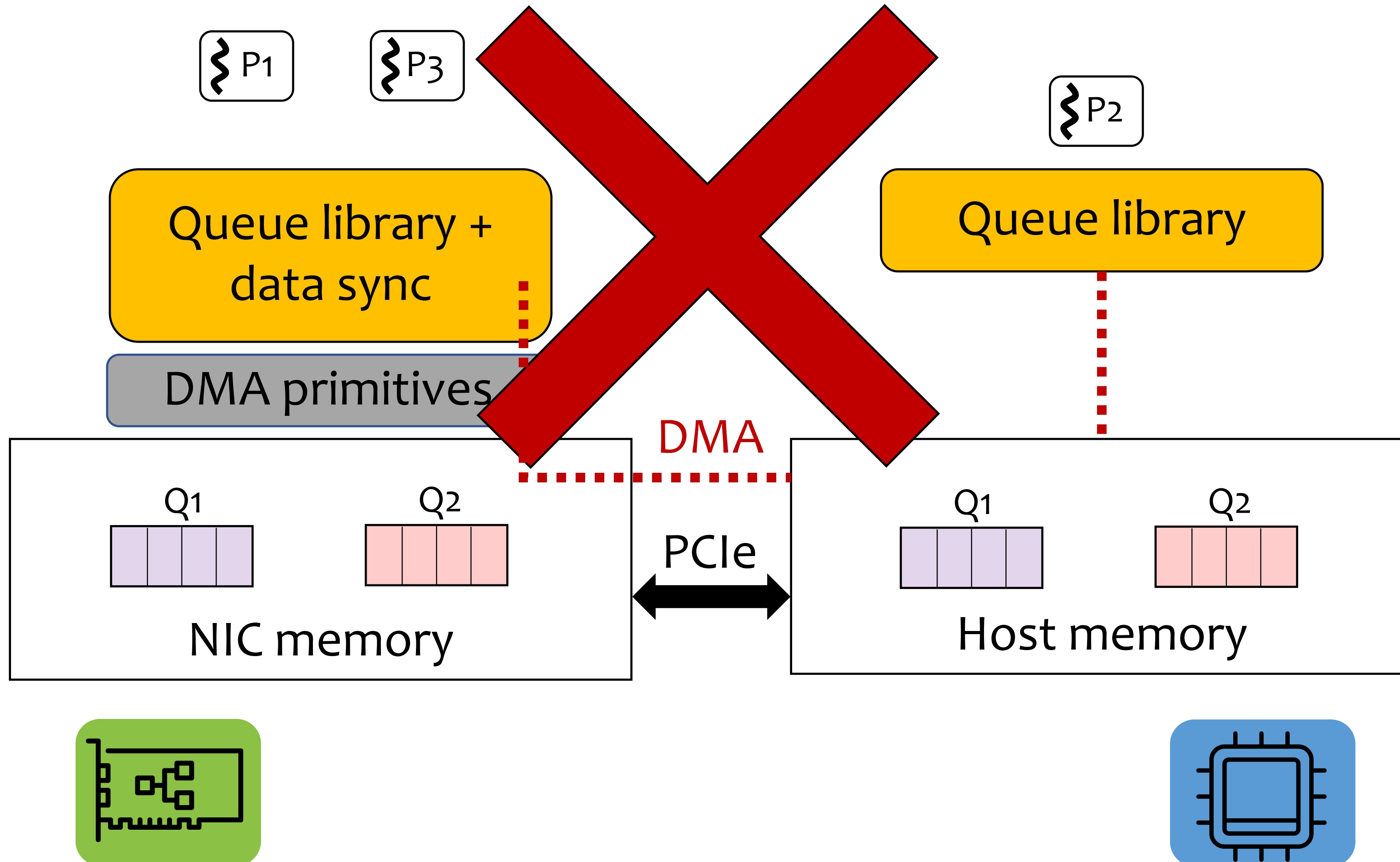





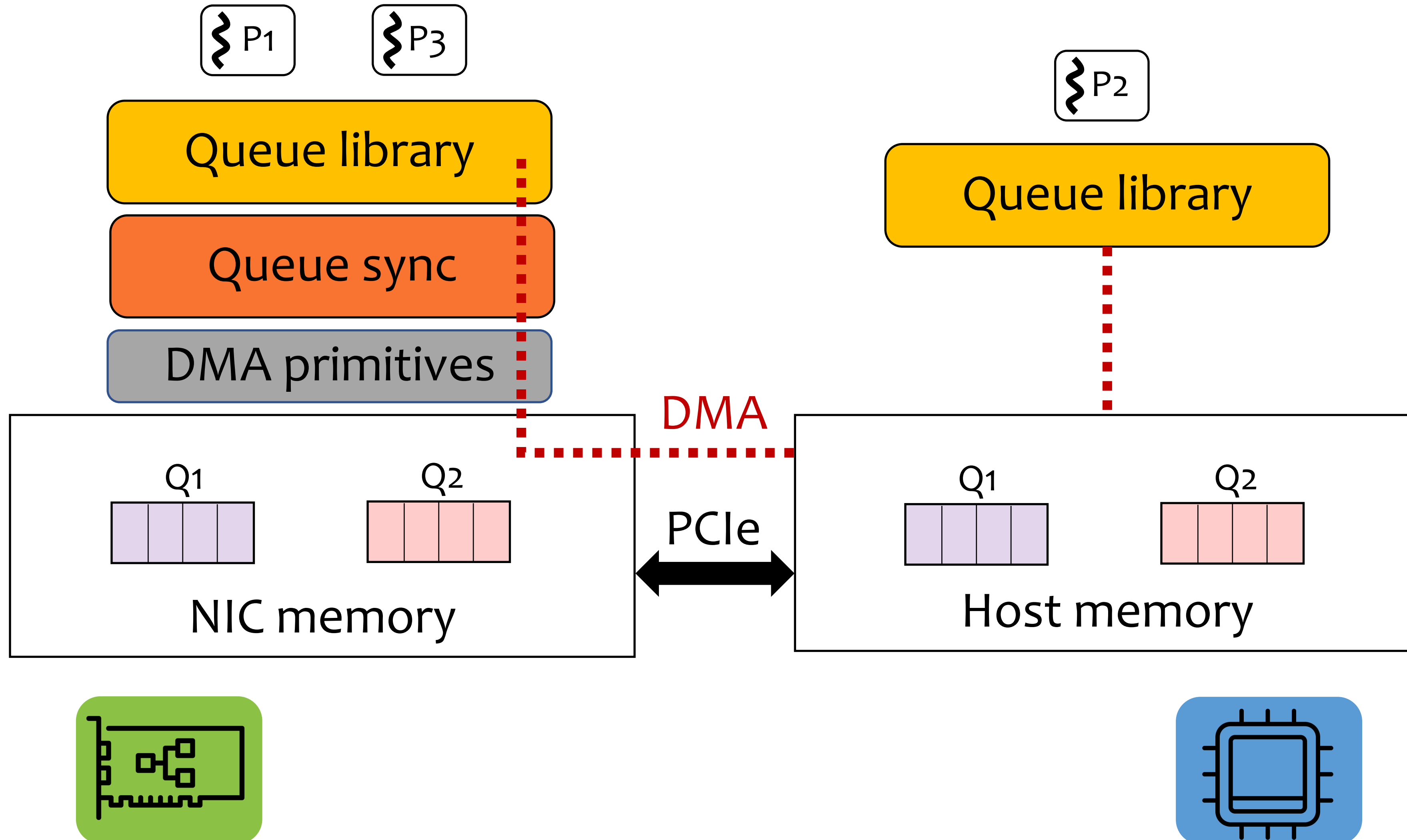
# Queue Implementation Challenge



# Queue Implementation Challenge



# Queue Implementation Challenge



# Evaluation

## Does Floem help programmers explore different offload designs?

# Server Setup

**With Smart NIC**

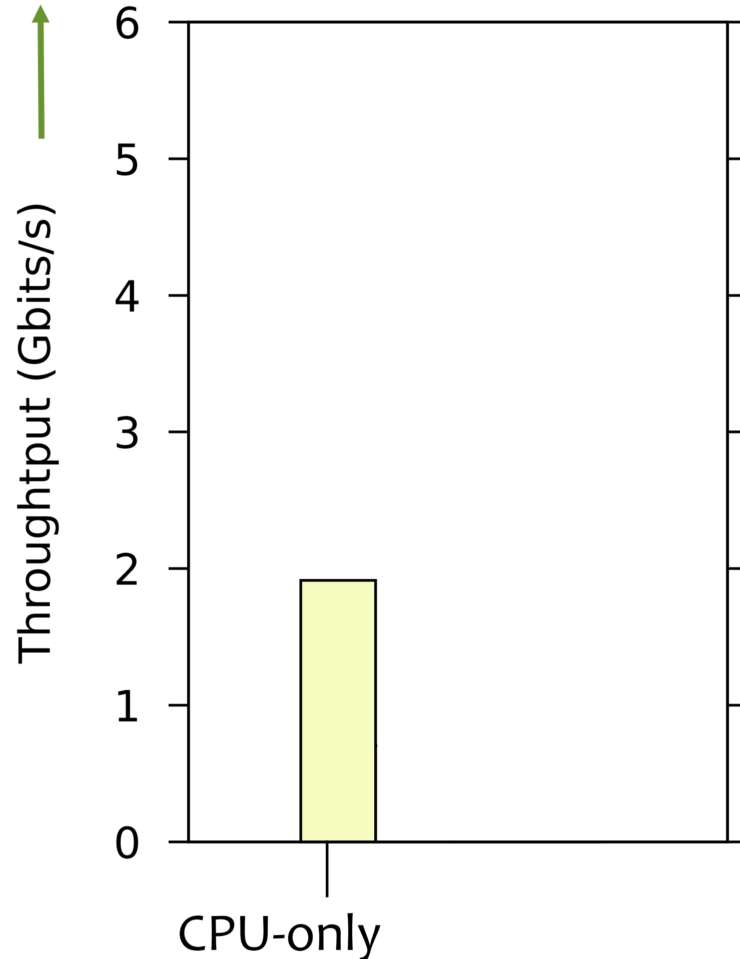
Cavium LiquidIO NIC
- two 10Gbps ports
- 12-core 1.20GHz cnMIPS64 processor
- 4GB memory

**Without Smart NIC**

Intel X710 NICs
- two 10Gbps ports
- DPDK (bypass OS networking stack)

6-core Intel X5650

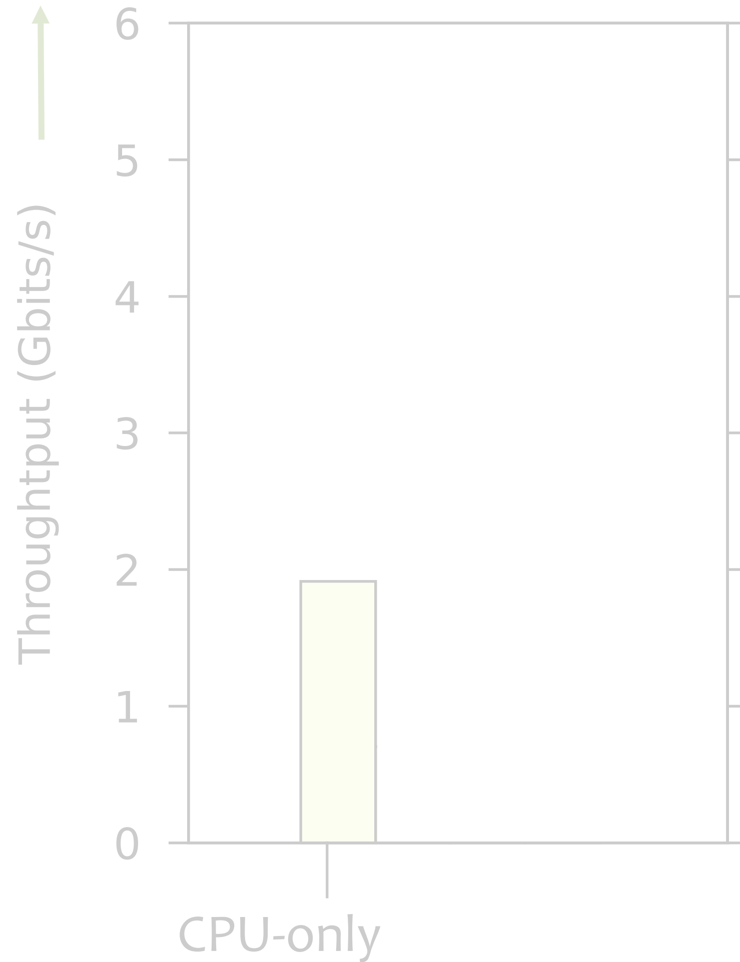
# Case Study: Key-Value Store




**Workload**

- 100,000 key-value pairs
- 32-byte keys and 64-byte values
- Zipf distribution (s = 0.9)
- 90% GET and 10% SET

# Case Study: Key-Value Store

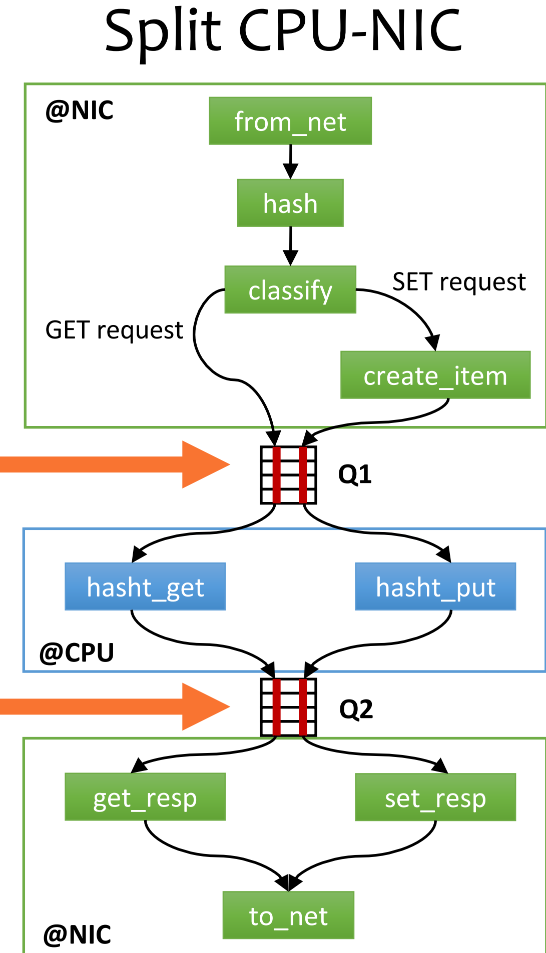



**Code relevant to communication**

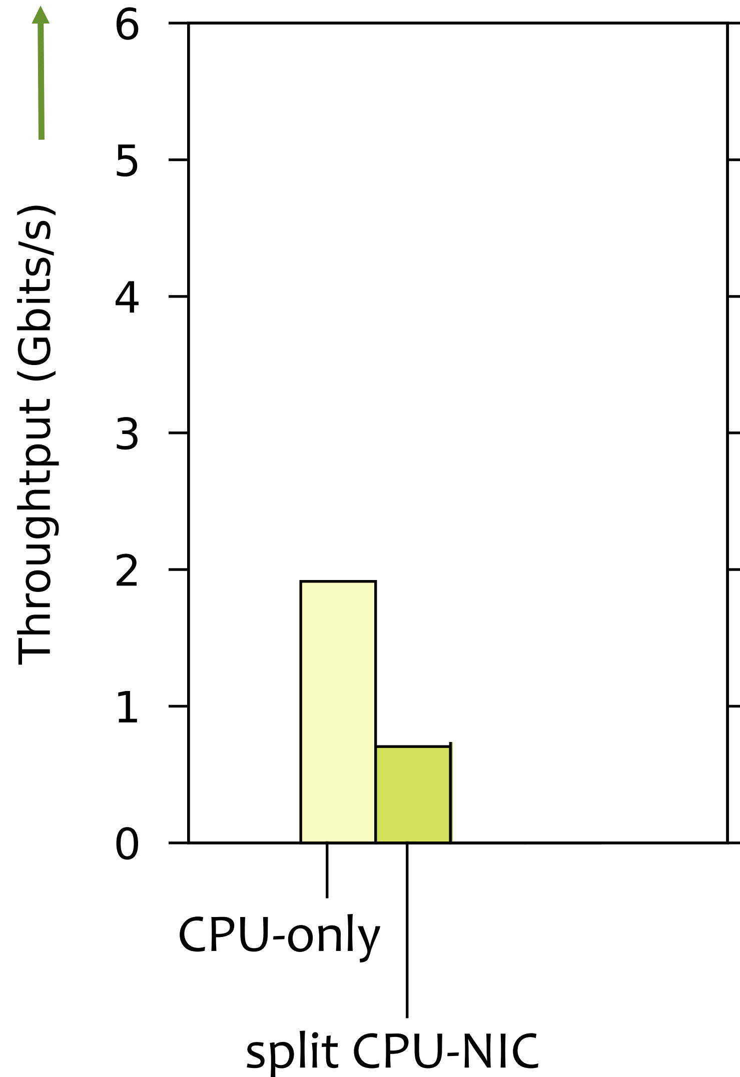
C program
+ 240 lines

Floem program
15 lines

Split CPU-NIC



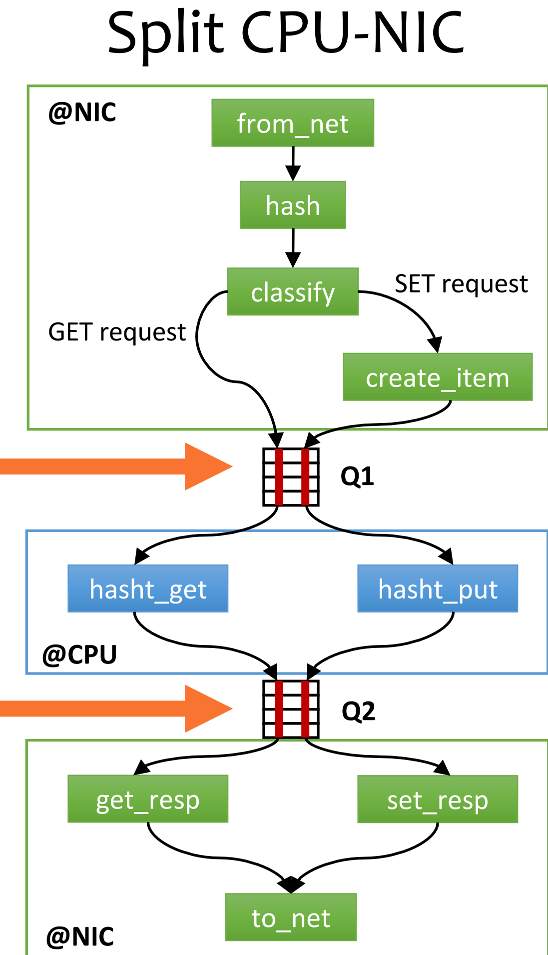
# Case Study: Key-Value Store



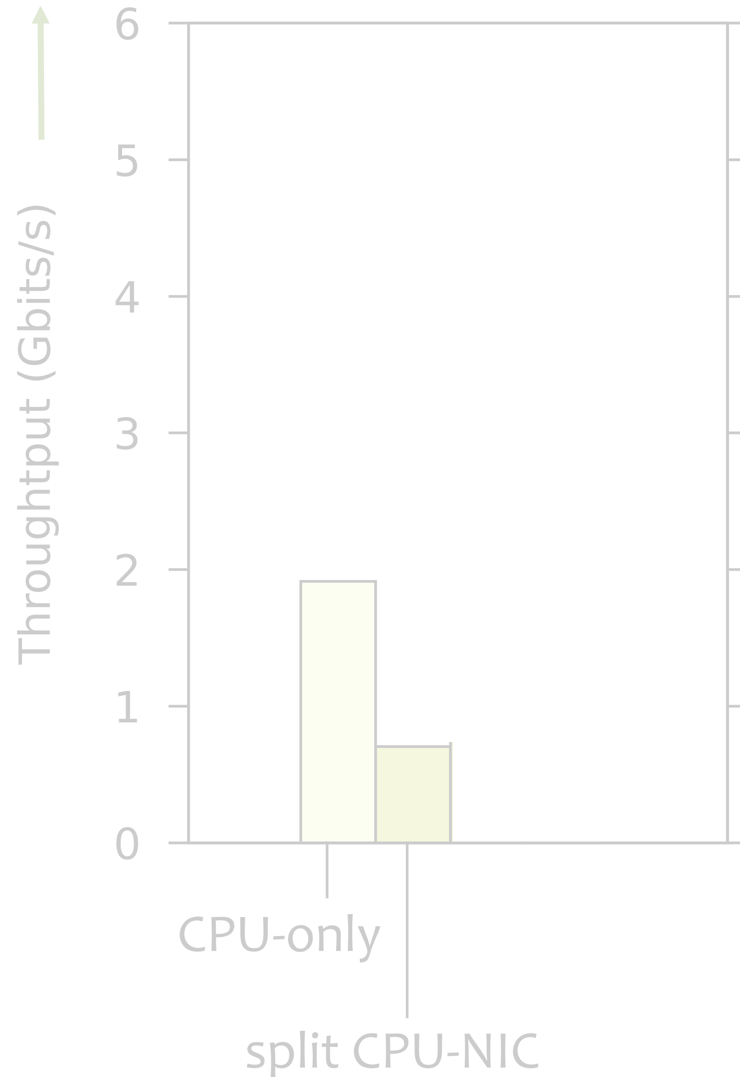

**Code relevant to communication**

C program
+ 240 lines

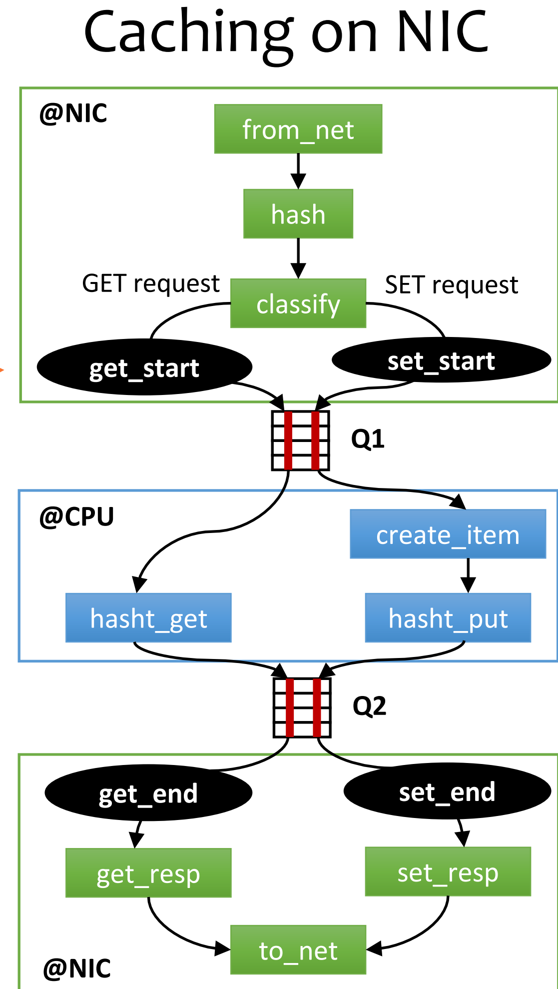
Floem program
15 lines



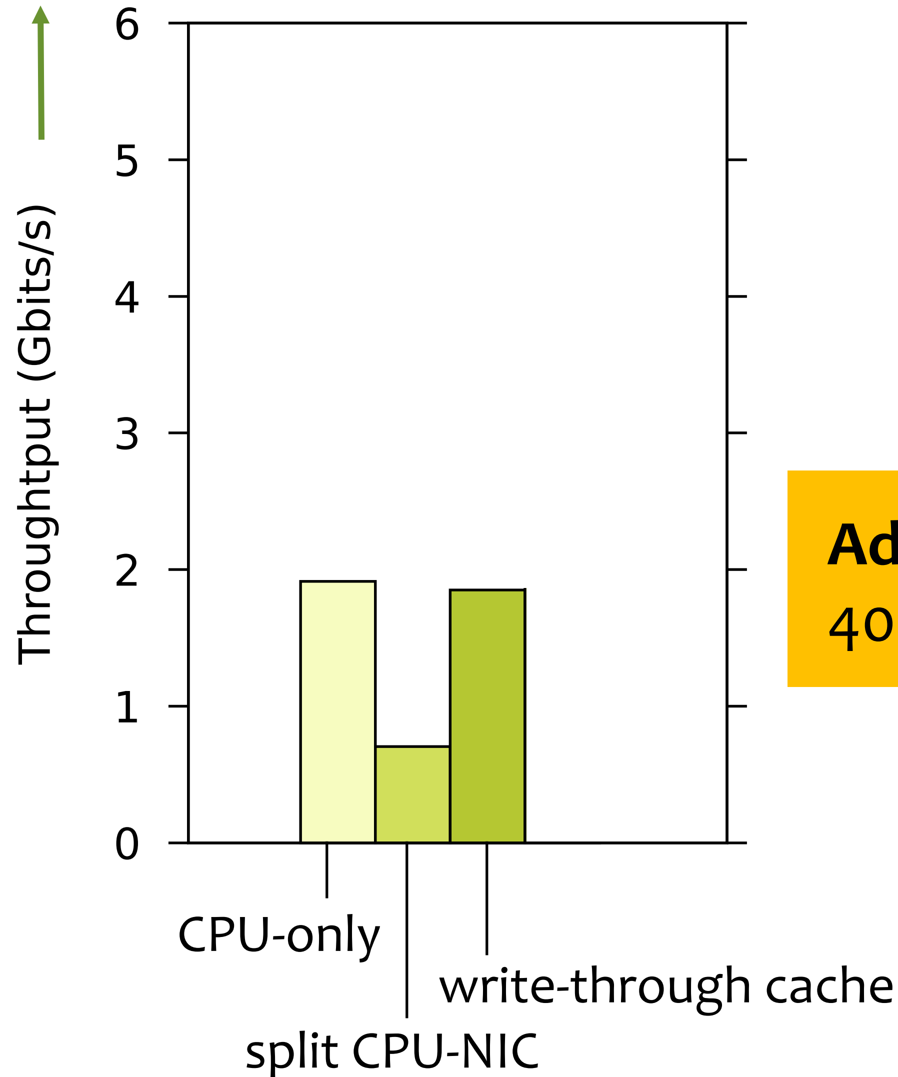
# Case Study: Key-Value Store



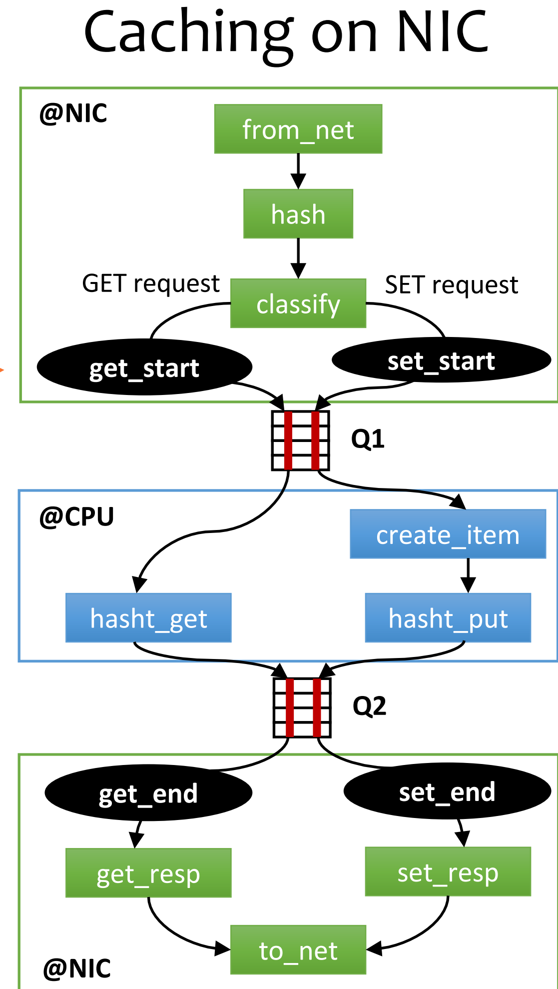

**Add cache in Floem**
40 lines



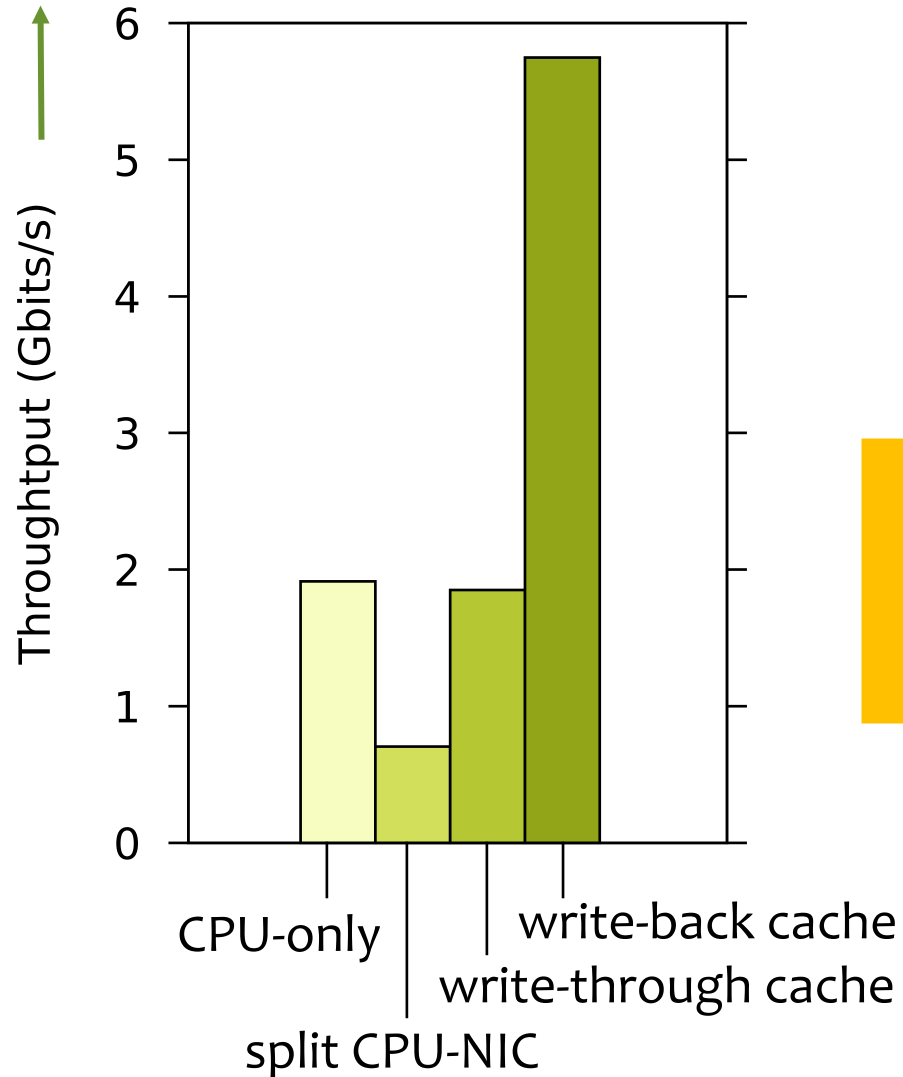
# Case Study: Key-Value Store




**Add cache in Floem**
40 lines

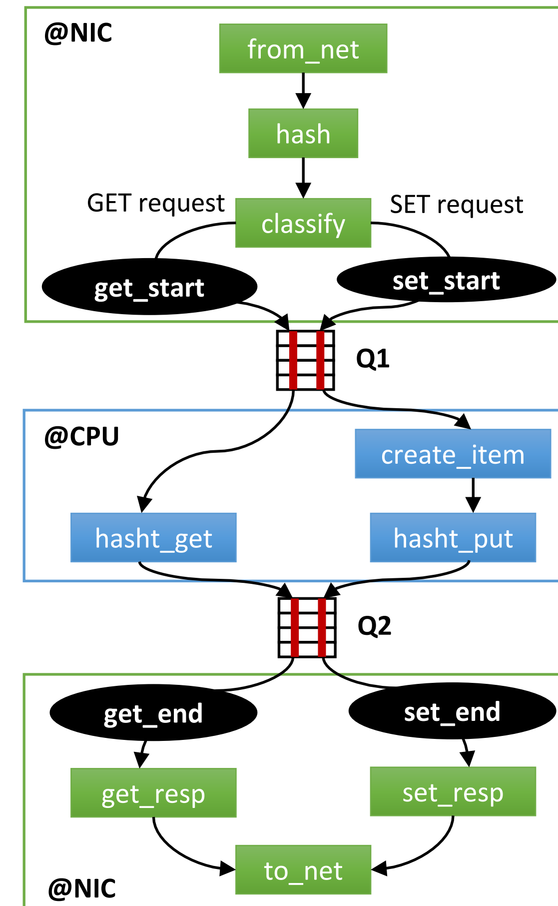


# Case Study: Key-Value Store




**Change policy parameter**
1 line

Caching on NIC



# Evaluation: Other Applications

## Distributed real-time data analytics (Storm)

- First offload: worse than CPU-only
- Second offload: 96% improvement with 23 lines of code

## Encryption

AES-CBC-128

## Flow classification

Use a count-min sketch on the header 5-tuple

## Network sequencer

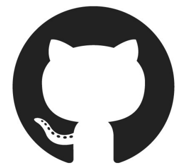
Use a group lock

# Conclusion

**Takeaway:** high-level programming abstractions

- control implementation strategies
- avoid low-level details

**Result:** minimal changes to explore different designs



github.com/mangpo/floem