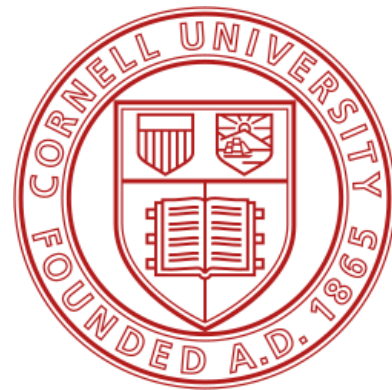




Data-plane programmability across heterogeneous networks

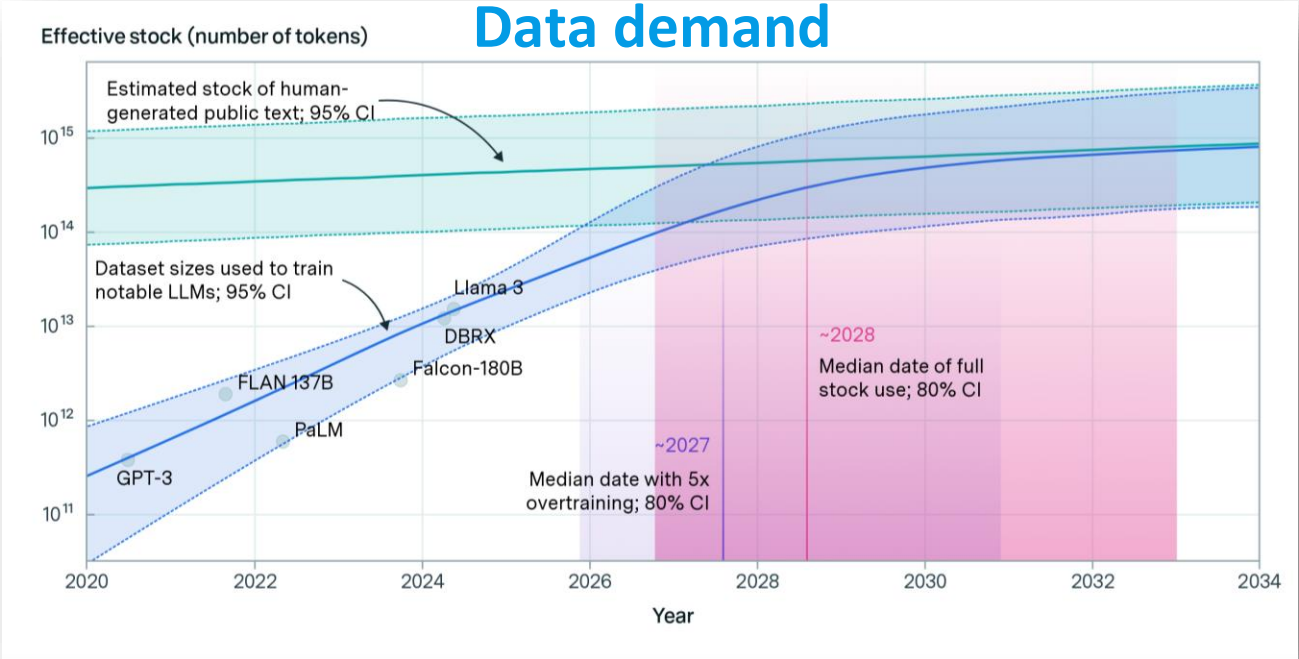
Jiaxin Lin

Assistant Professor, Electrical and Computer Engineering

<https://www.csl.cornell.edu/~jiaxinl>

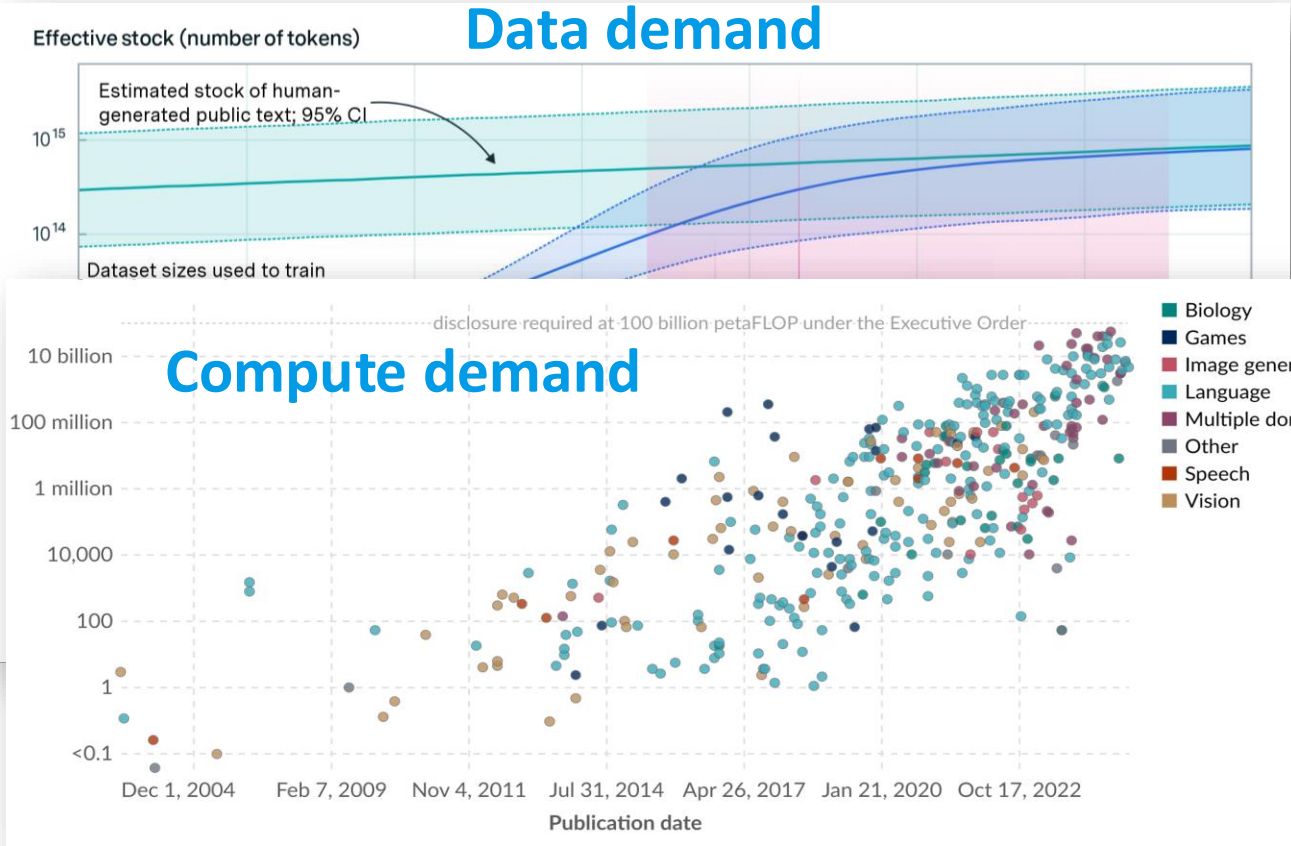


OpenAI



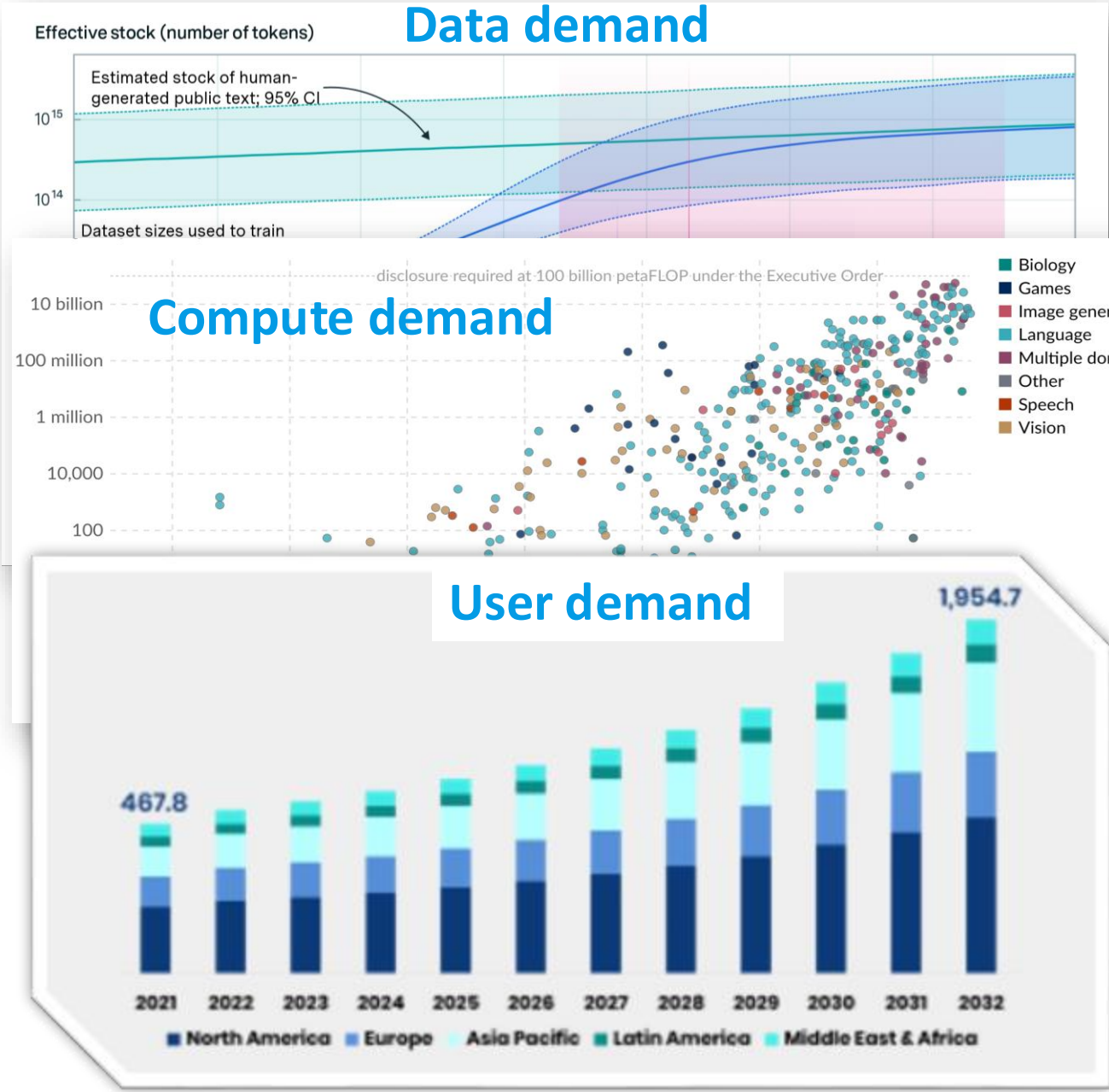


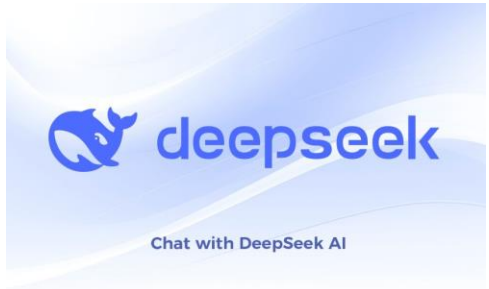
OpenAI



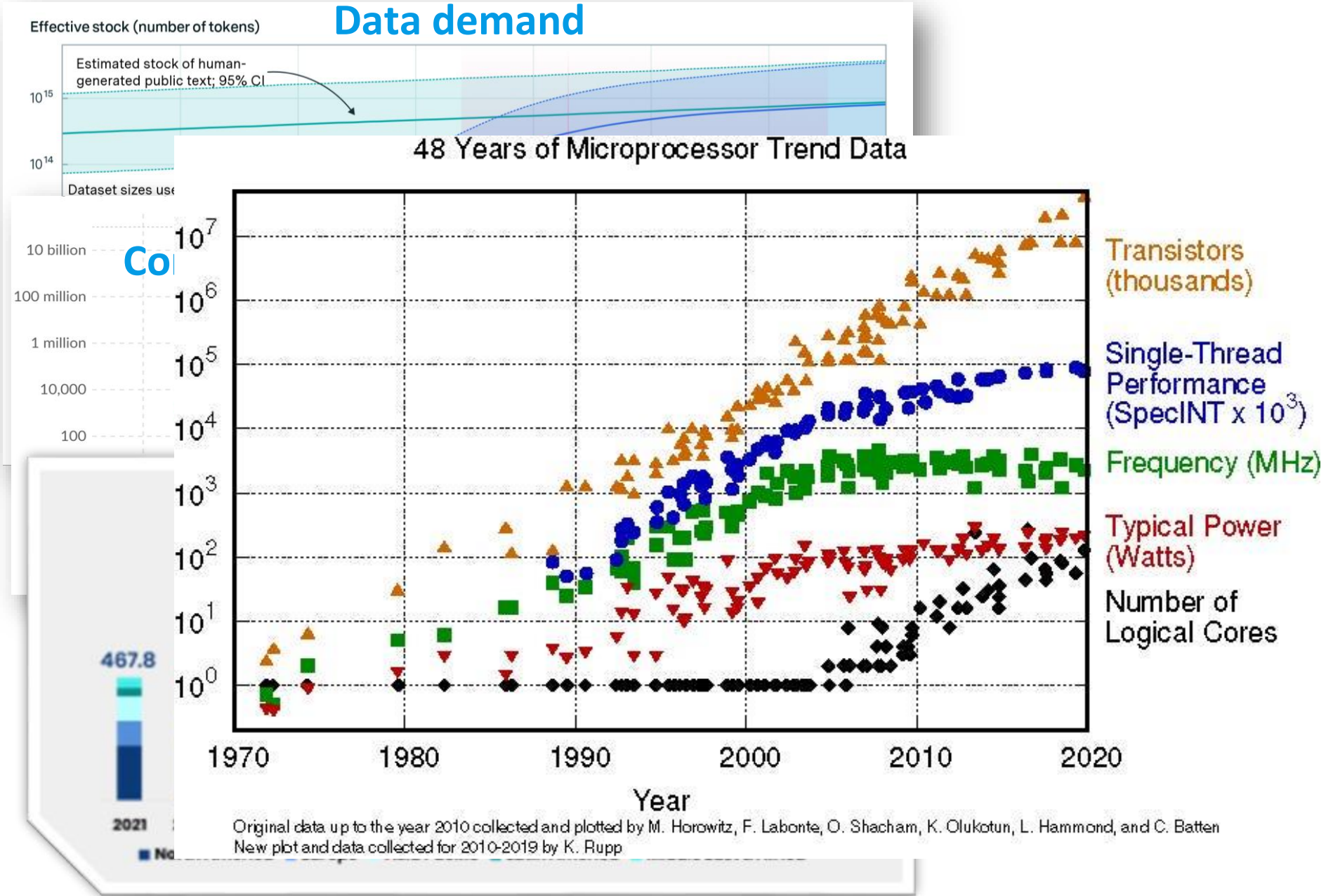


OpenAI

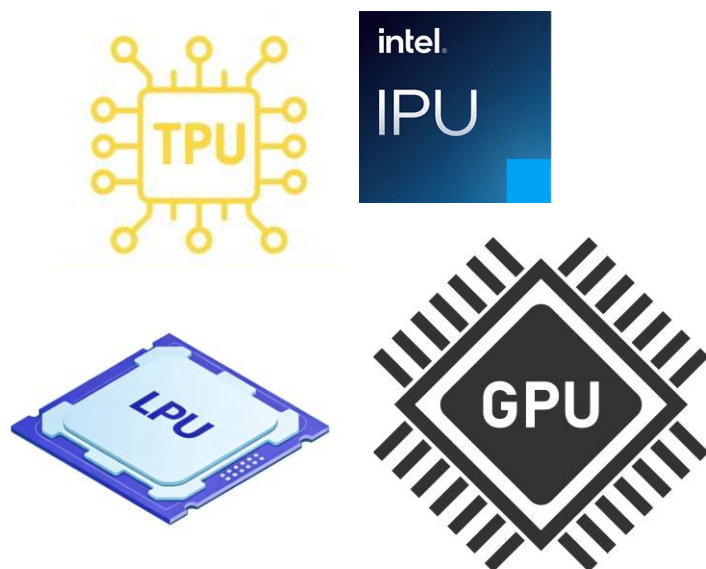




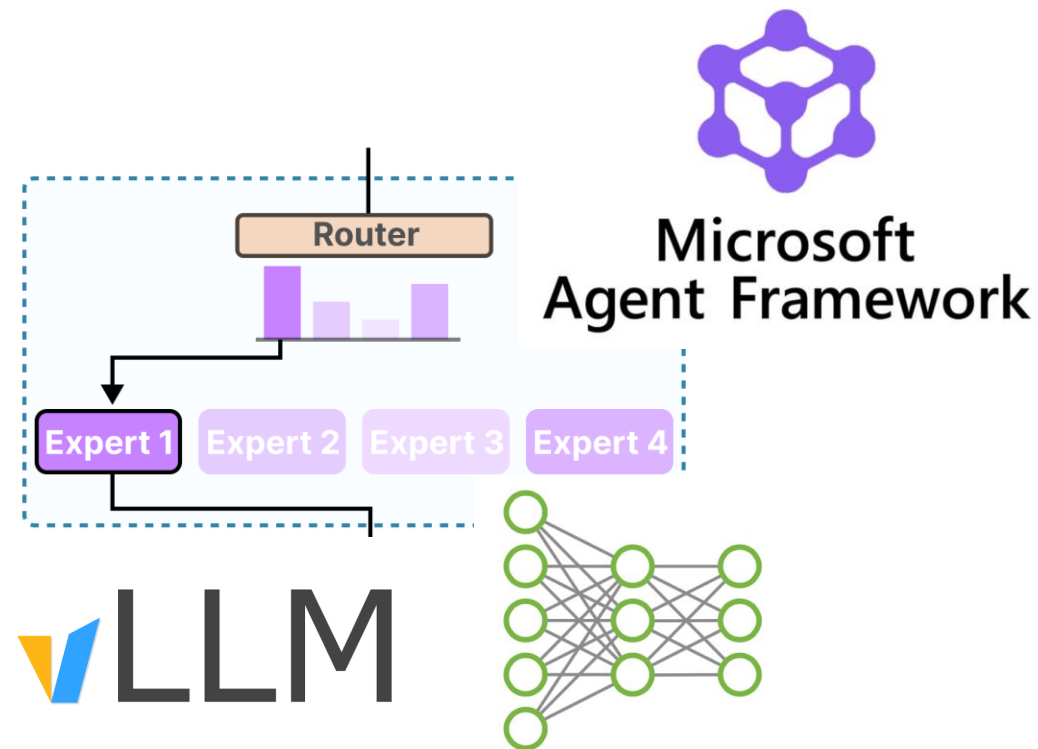
OpenAI



Heterogeneity and specialization help computing keep up



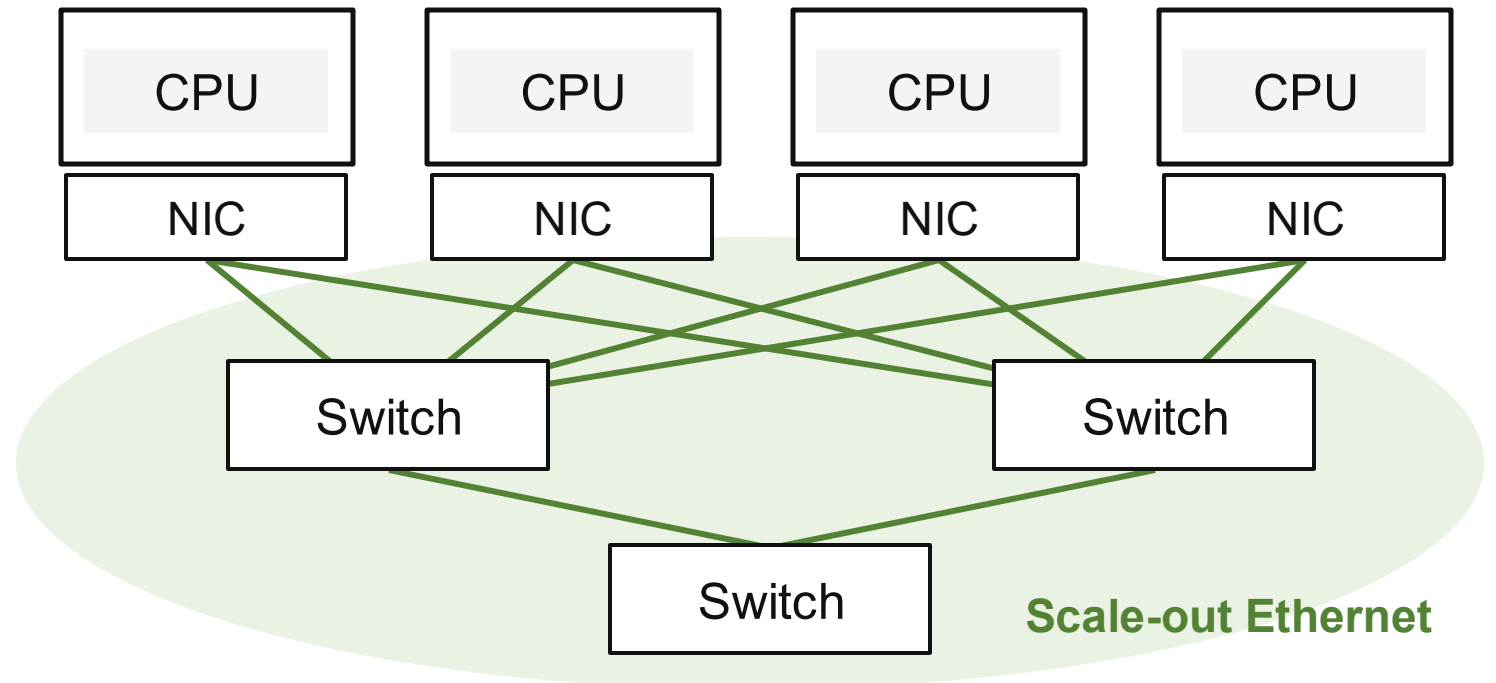
Compute specialization



System specialization

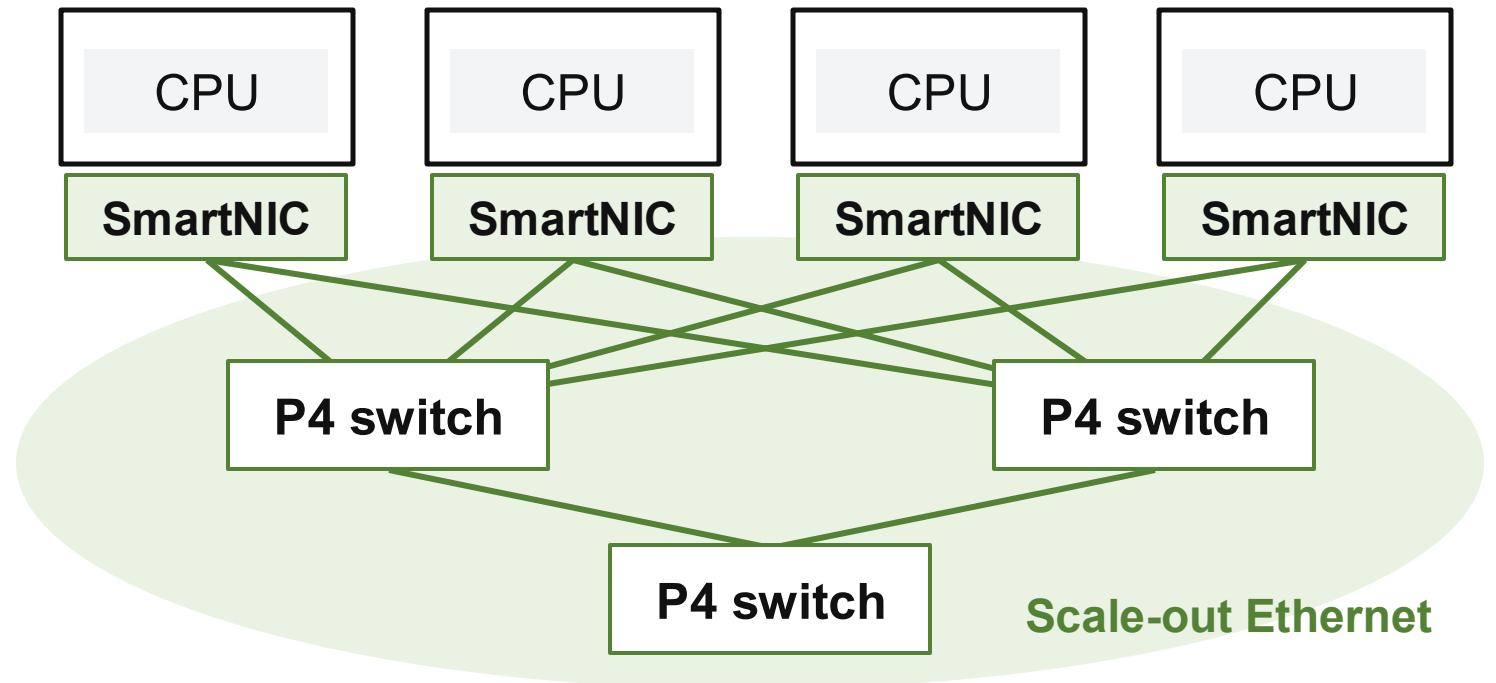
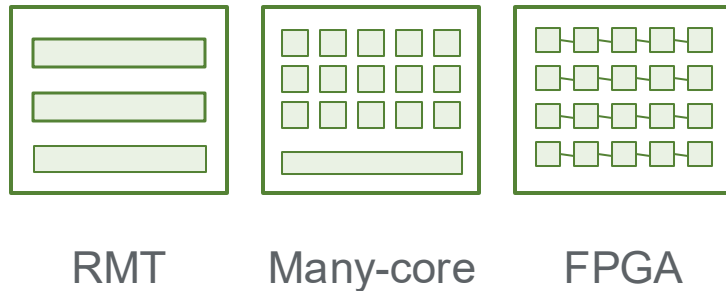
Programmable networks inherit this trend

- Network device heterogeneity
 - **Diverse architectures and execution models**



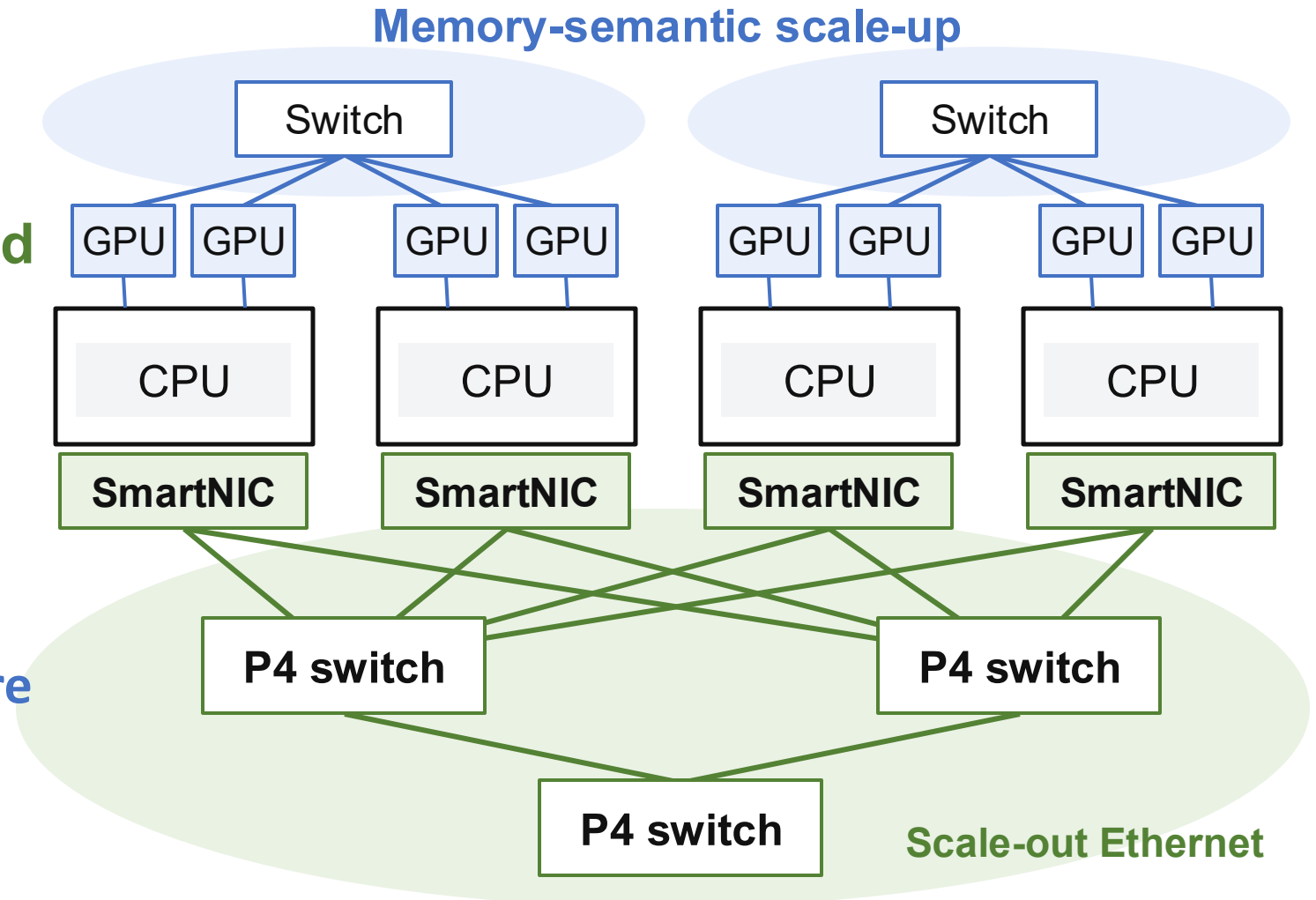
Programmable networks inherit this trend

- Network device heterogeneity
 - **Diverse architectures and execution models**



Programmable networks inherit this trend

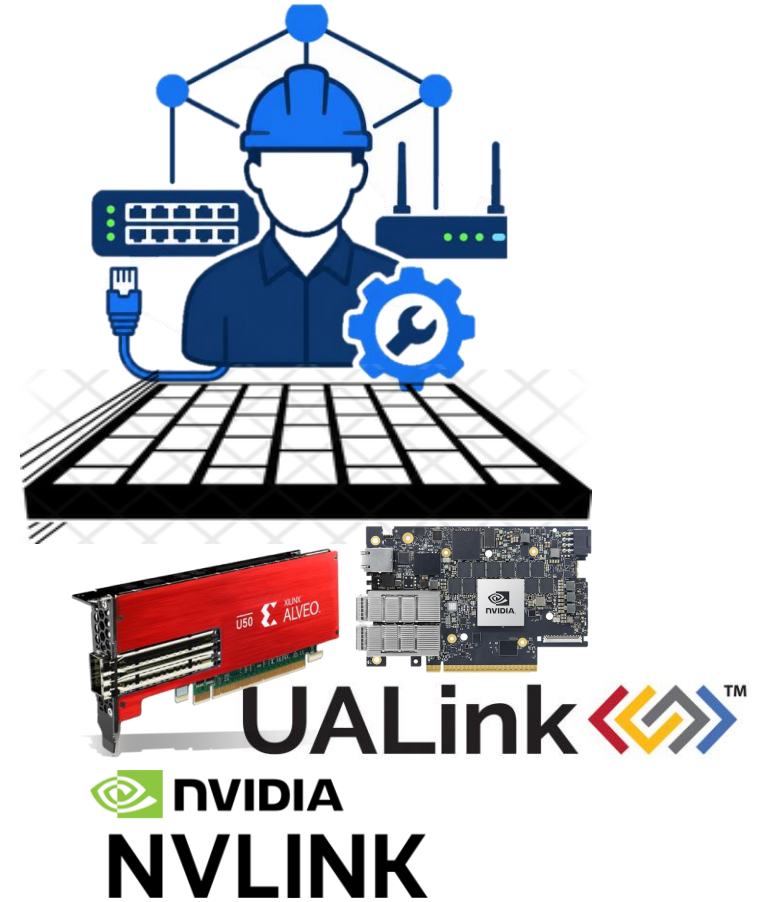
- Network device heterogeneity
 - **Diverse architectures and execution models**
- Communication protocol specialization
 - **Memory-semantic load/store protocols**



Programmability challenges under heterogeneity

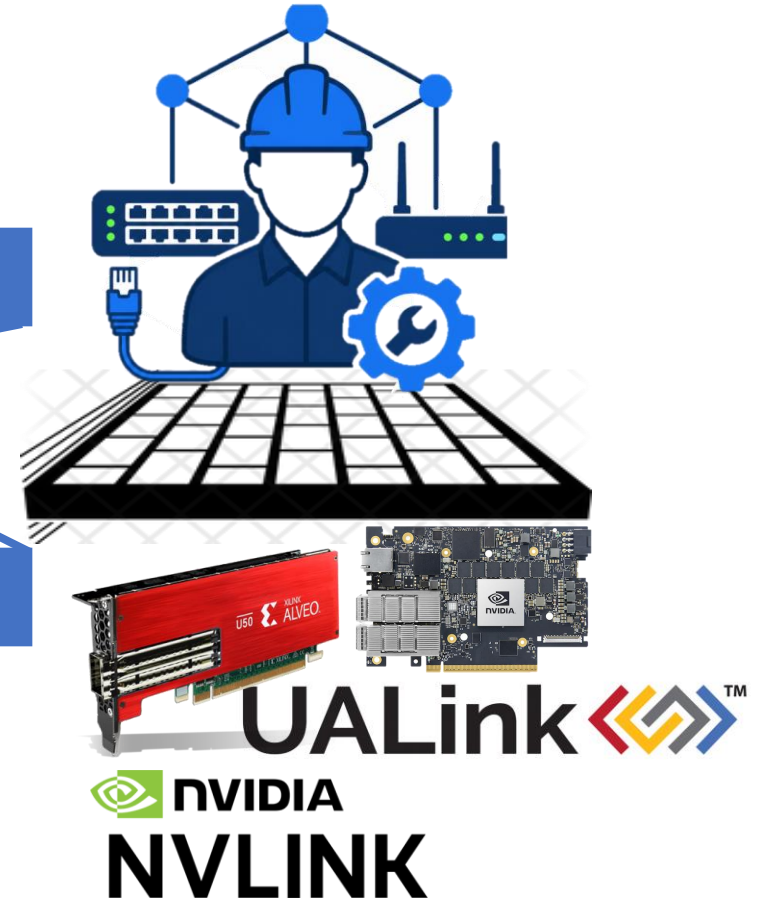
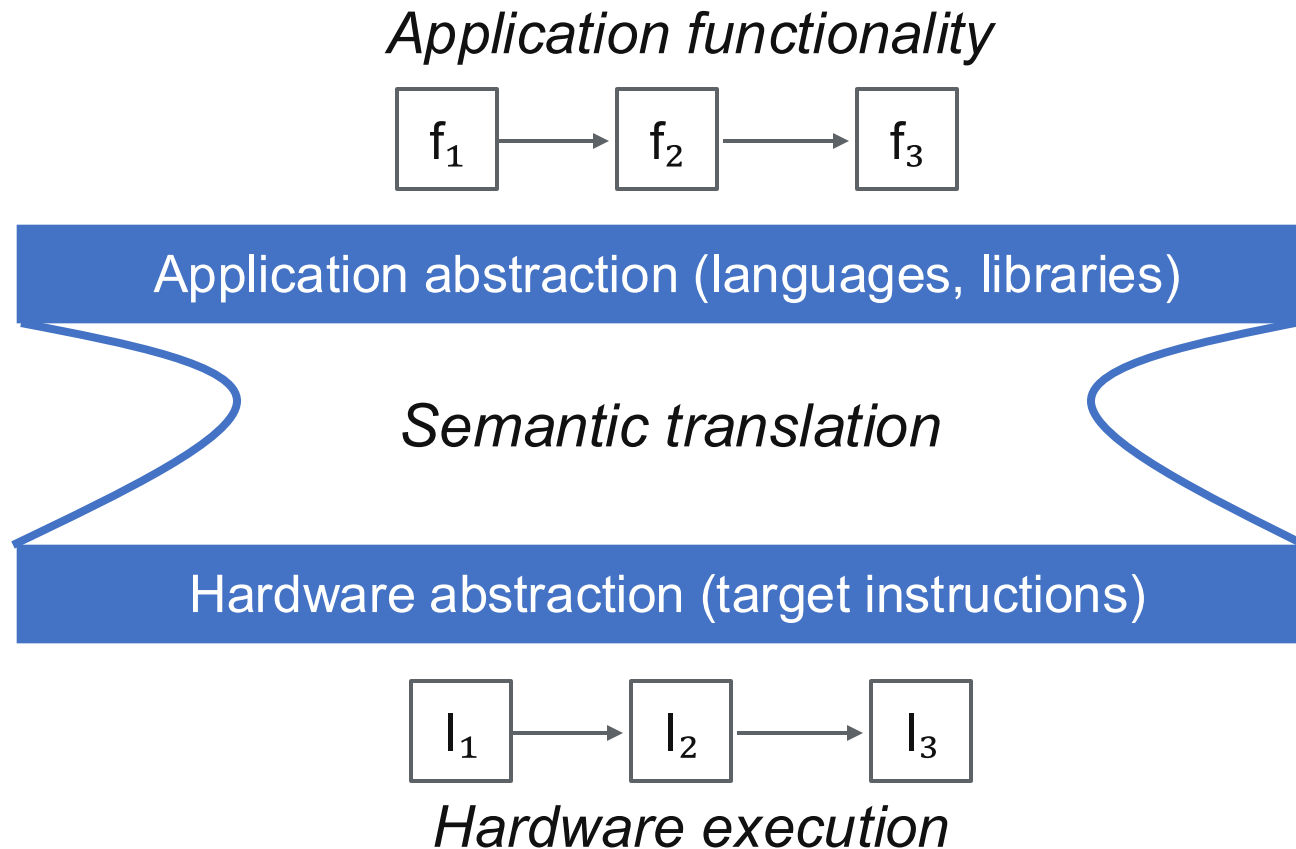
Network programmers must manage

- Specialized architectures
- Target-specific optimizations
- Performance and resource limits



How can we separate these concerns?

Abstraction and semantic translation



P4 provides this separation for packet-processing pipelines.

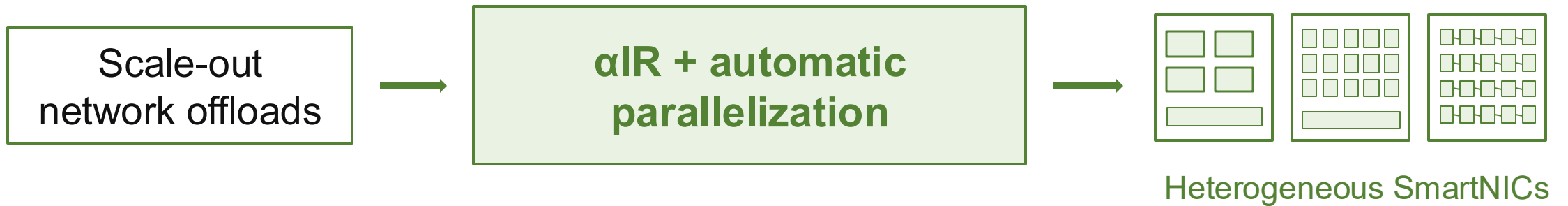
- Can we bridge semantic gaps across diverse devices and interconnects?

Bridging programming and hardware execution

Alkali

NSDI '25

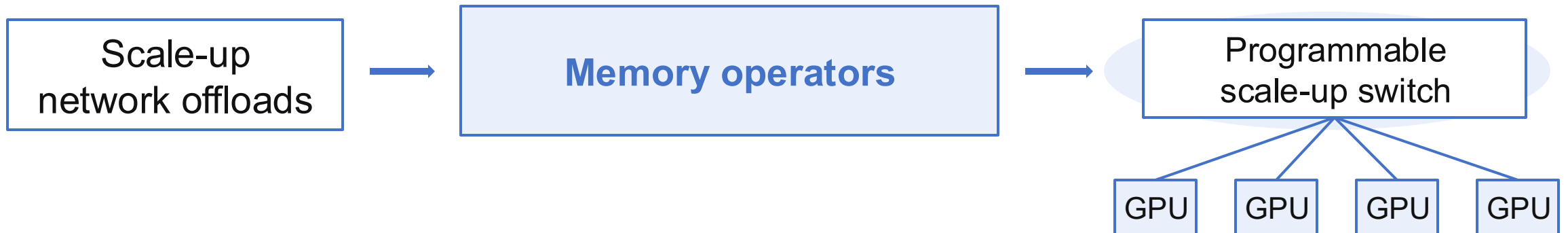
Portable compiler across SmartNICs

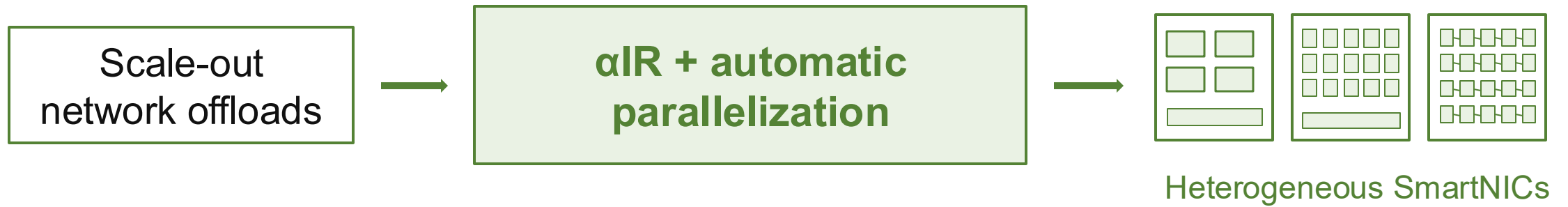


MemOps

HotNets '26

Programmability for memory transactions

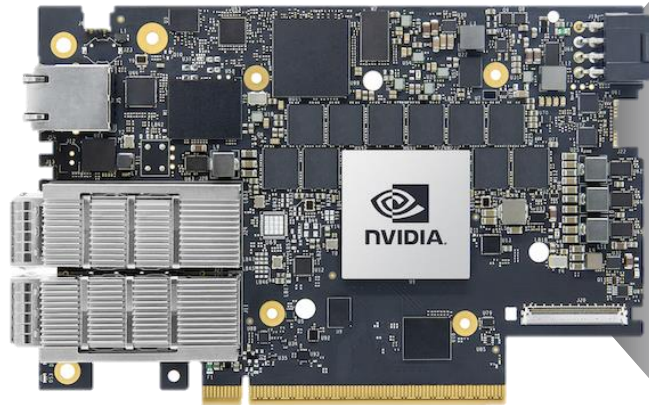




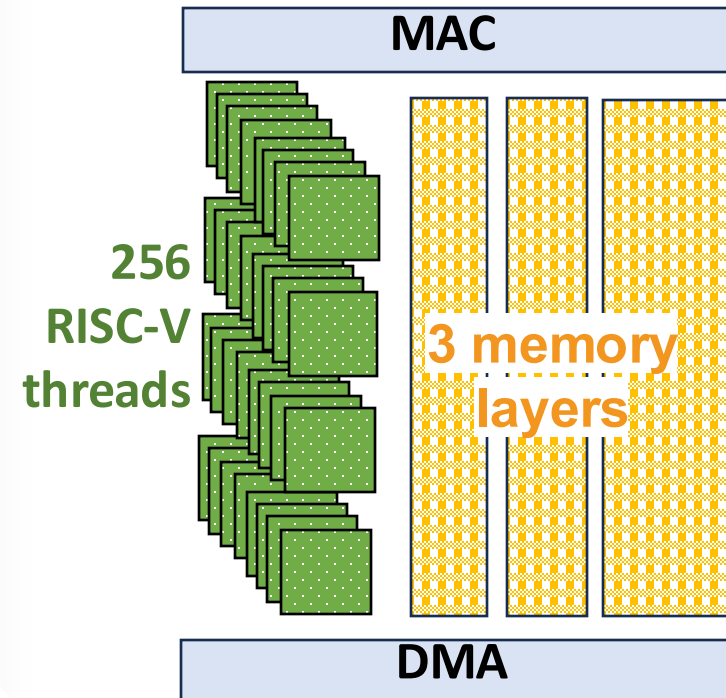
Alkali: portable and high-performance NIC programming

NSDI '25

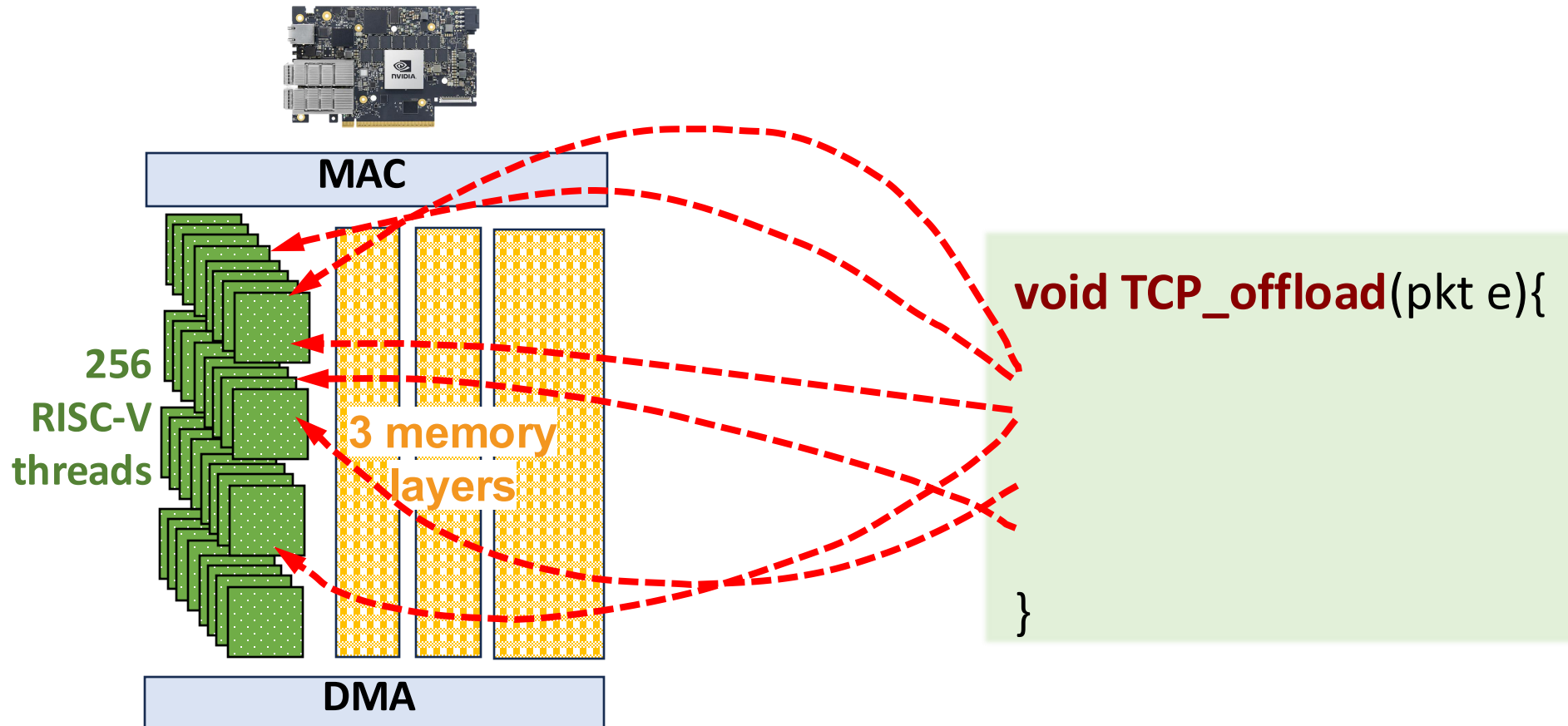
Rich hardware parallelism in SmartNICs



NVIDIA BlueField-3

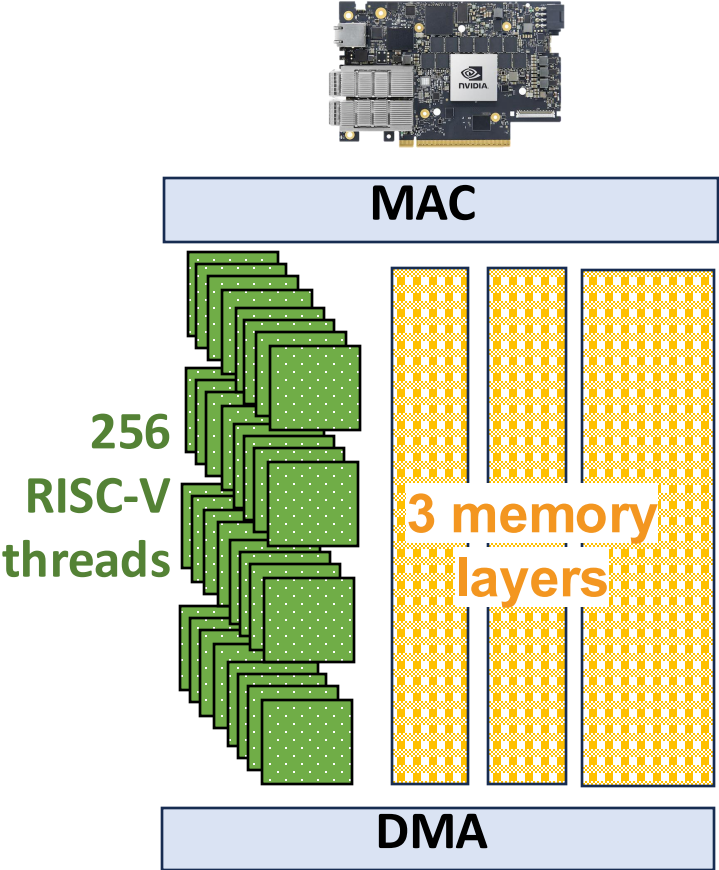


Barrier 1: low-level parallel programming

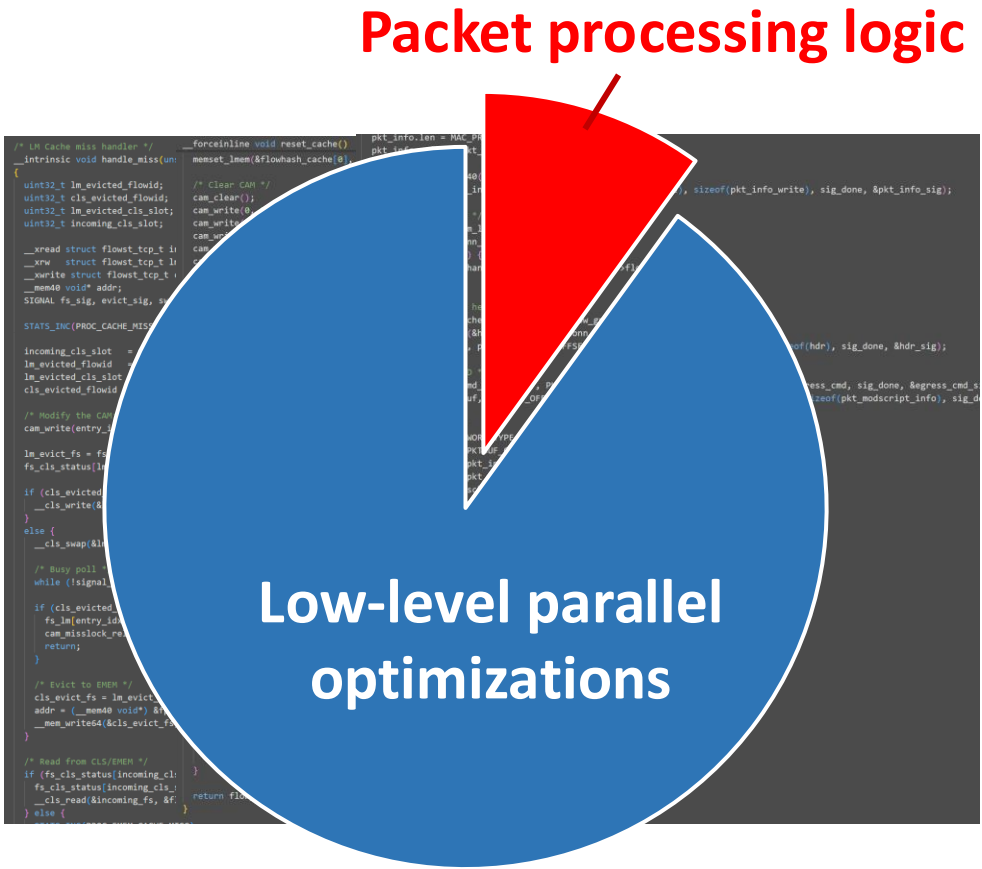


Low-level interfaces:
SDK with **500+** MicroC library functions

Barrier 1: low-level parallel programming

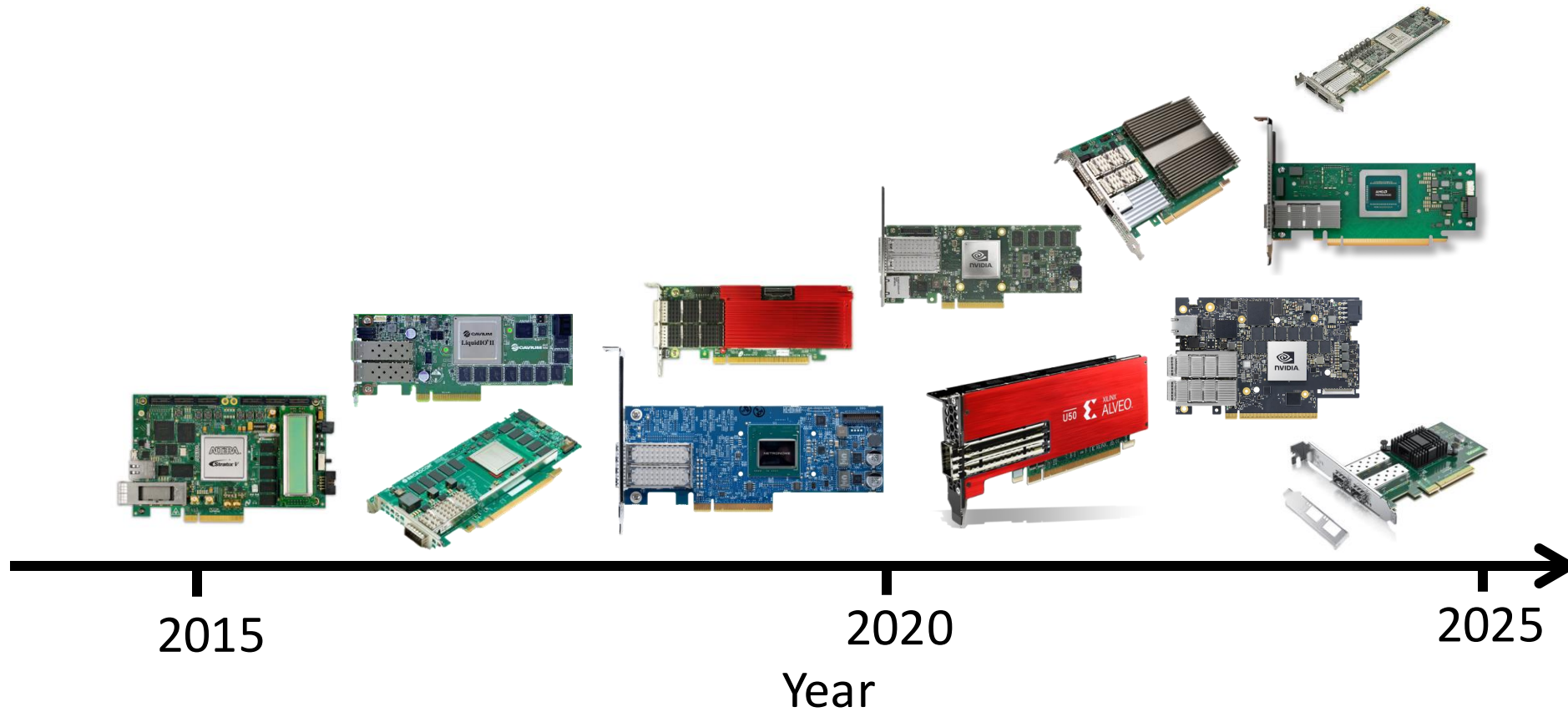


Low-level interfaces:
SDK with **500+** MicroC library functions

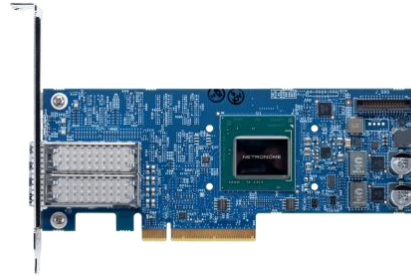


Distribution of lines of code

Barrier 2: non-portability



Barrier 2: non-portability



Agilio



FPGA



BlueField-2

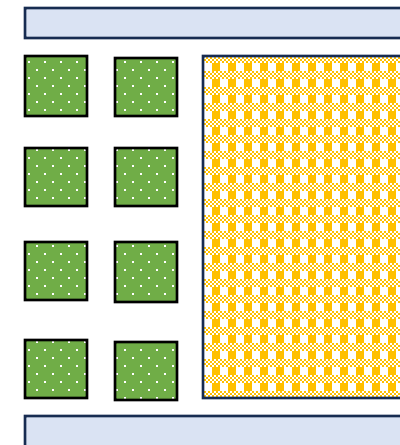
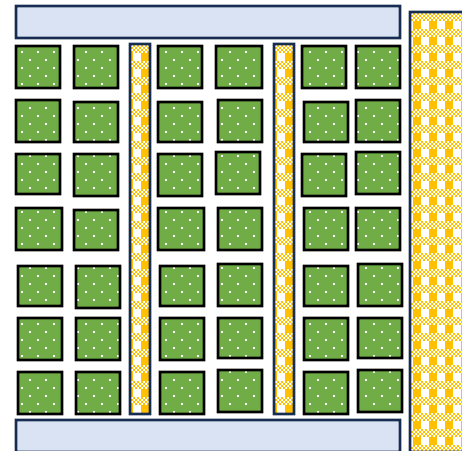
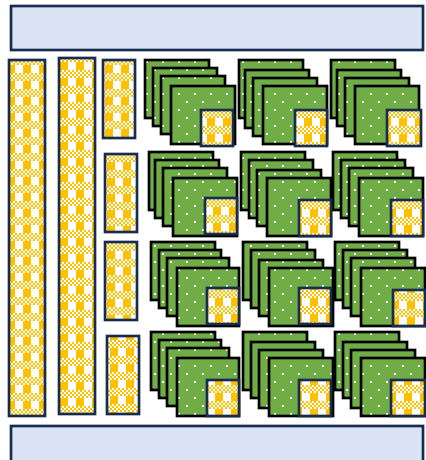
Interfaces:

MicroC with macros

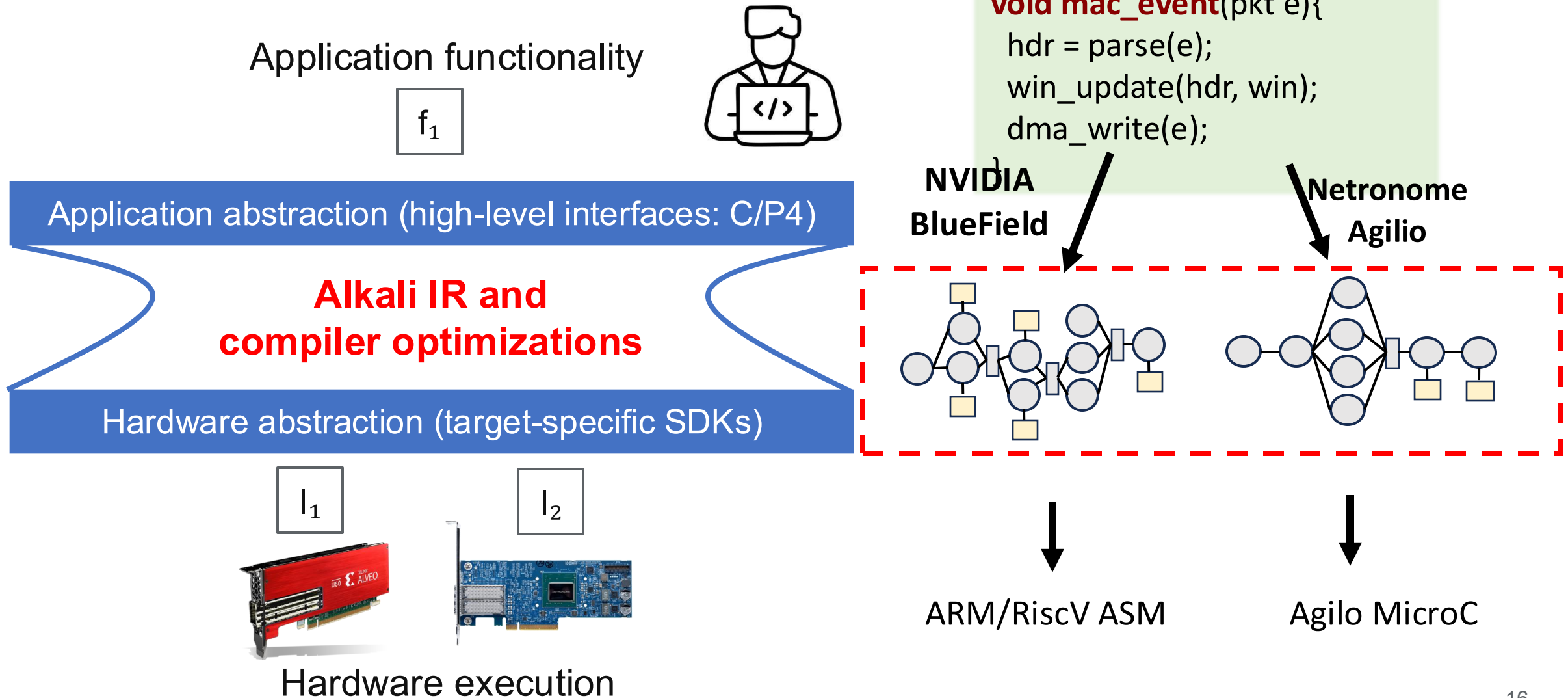
Verilog (RTL)

DPDK + extern libc

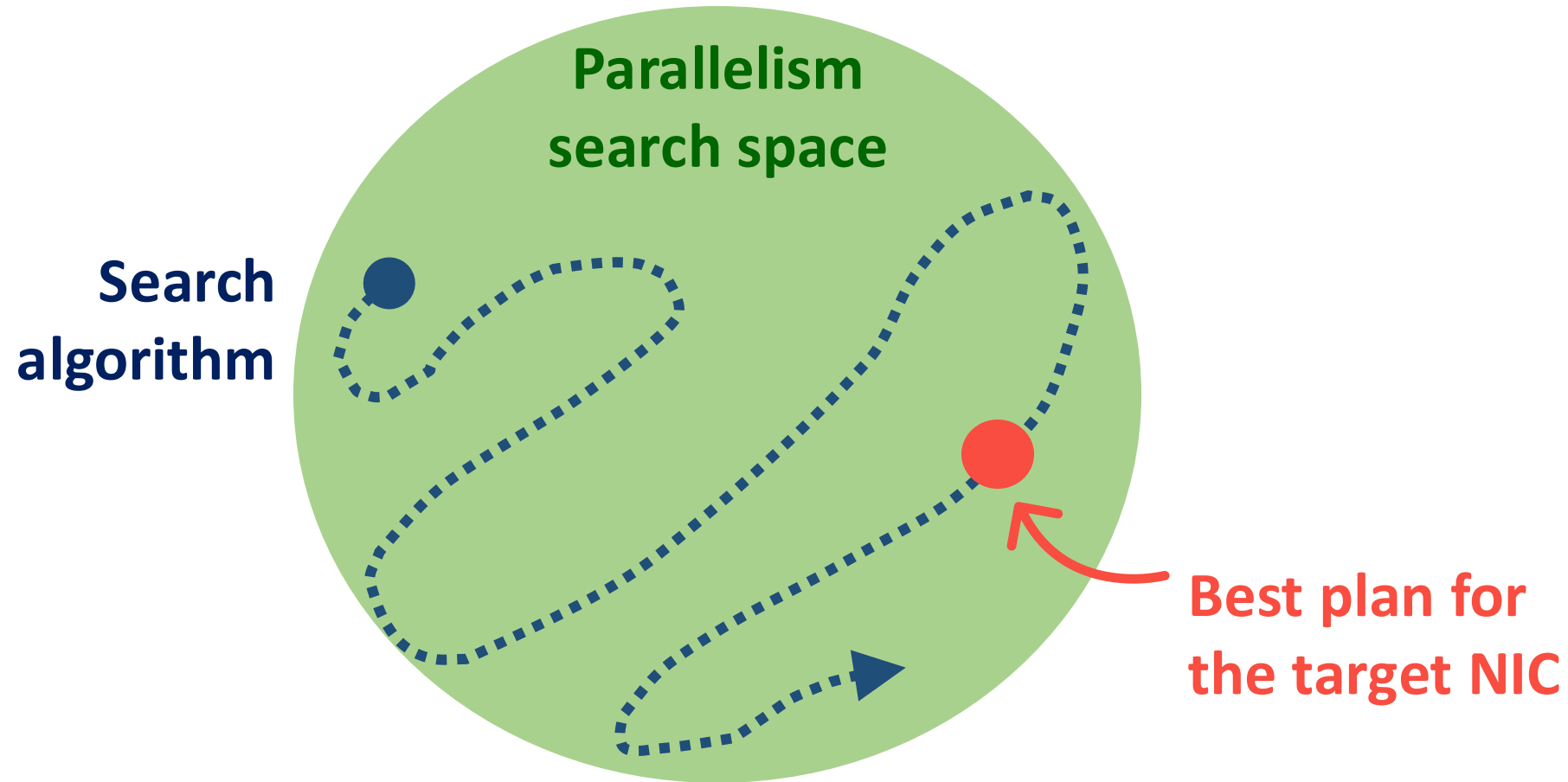
Architectures:



Alkali's compilation toolchain



Automatic parallelization: problem formulation

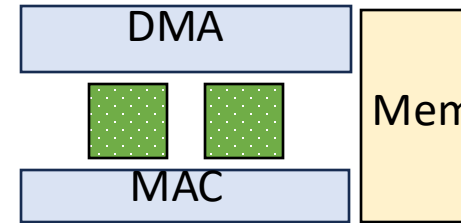


Defining the search space: parallelism types

```
Table T;  // flow table  
void mac_event(p){  
 ip = parse_ip(p);  
 c = T.lookup(ip);  
 T.update(ip, c + 1);  
}
```



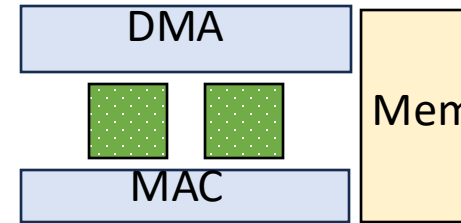
Target architecture



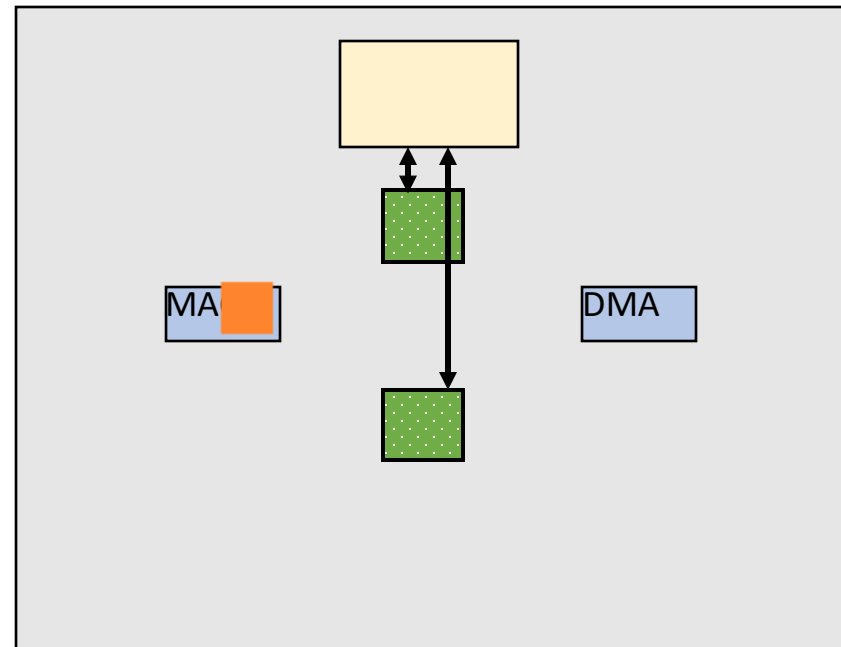
Parallelism type 1: packet parallelism

```
Table T;  // flow table  
void mac_event(p){  
 ip = parse_ip(p);  
 c = T.lookup(ip);  
 T.update(ip, c + 1);  
}
```

Target architecture



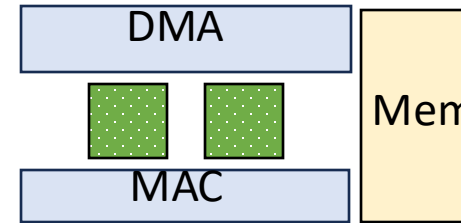
Packet
parallelism



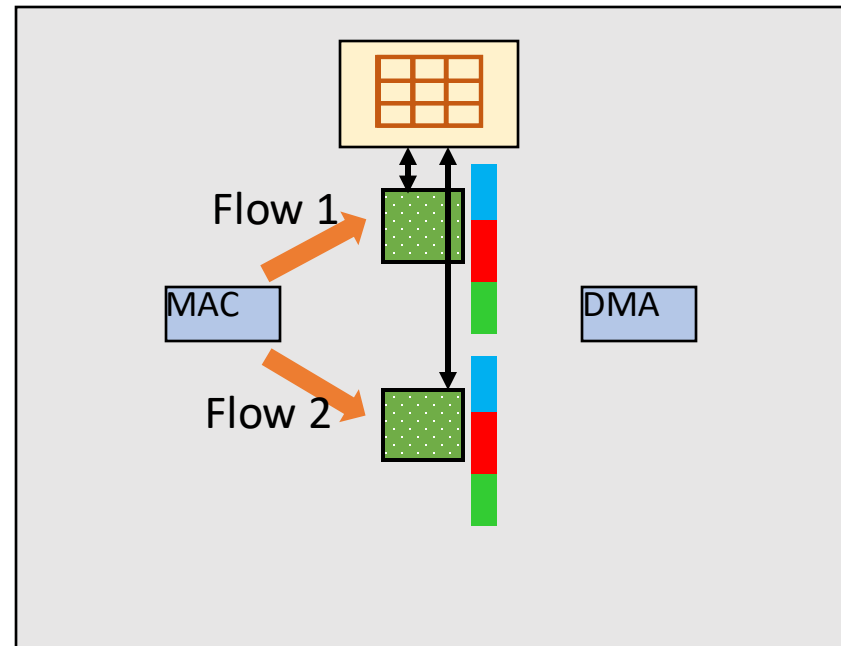
Parallelism type 2: flow parallelism

```
Table T;  // flow table  
void mac_event(p){  
   ip = parse_ip(p);  
   c = T.lookup(ip);  
   T.update(ip, c + 1);  
}
```

Target architecture



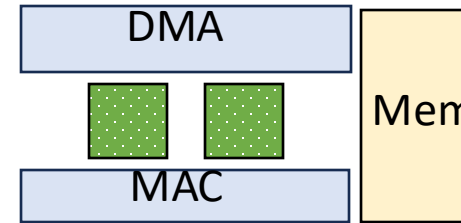
Flow
parallelism



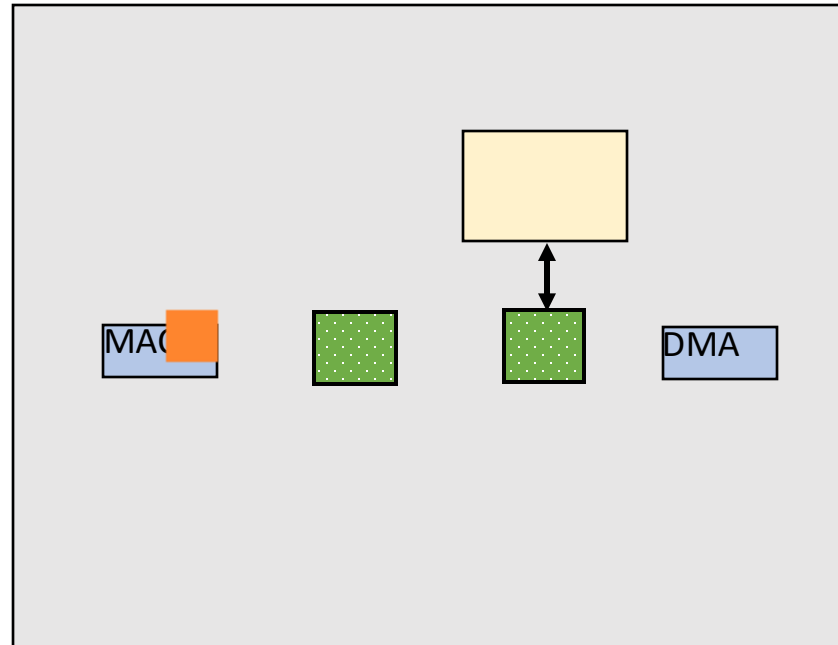
Parallelism type 3: pipeline parallelism

```
Table T;  // flow table  
void mac_event(p){  
 ip = parse_ip(p);  
 c = T.lookup(ip);  
 T.update(ip, c + 1);  
}
```

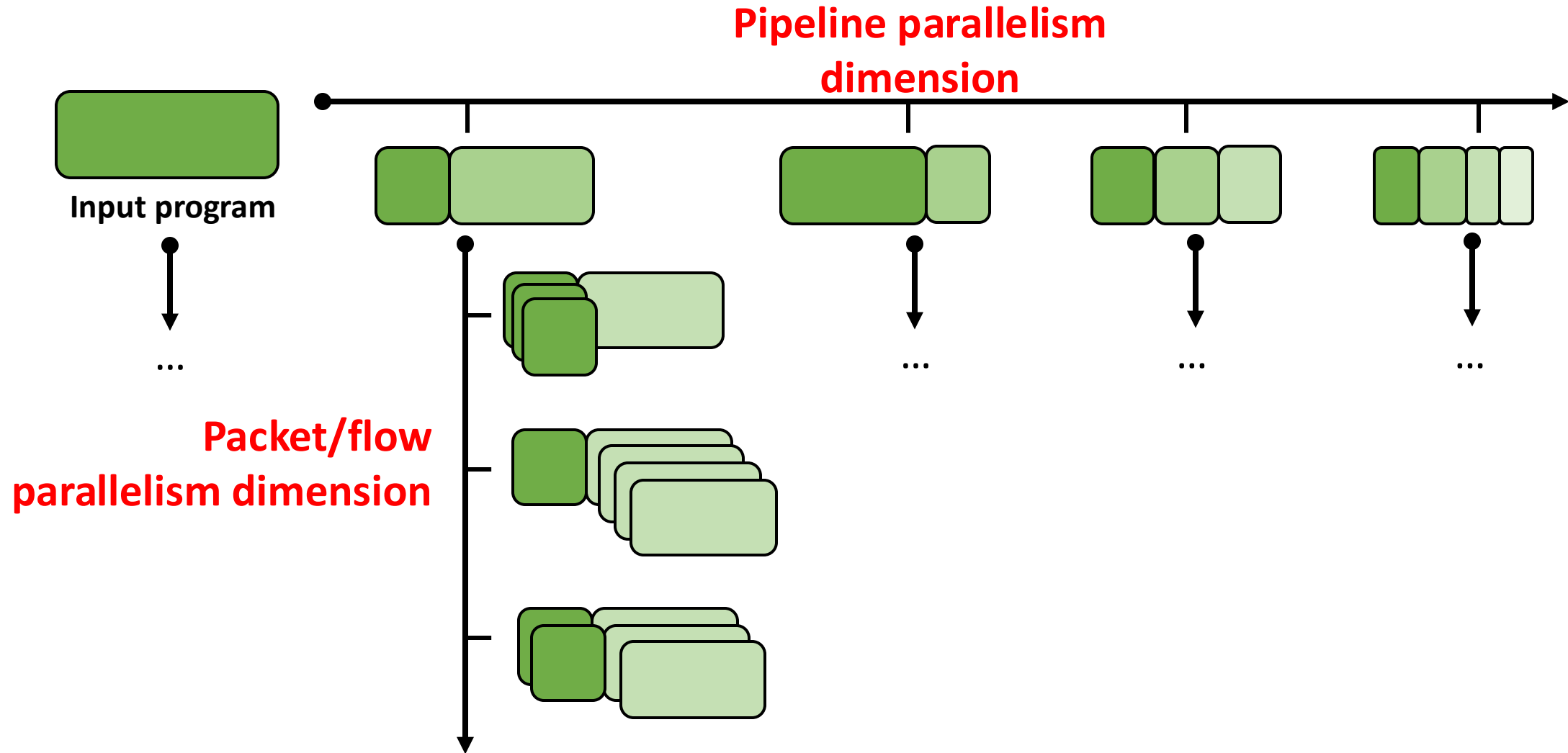
Target architecture



Pipeline
parallelism



Search space: pipeline, packet, and flow parallelism

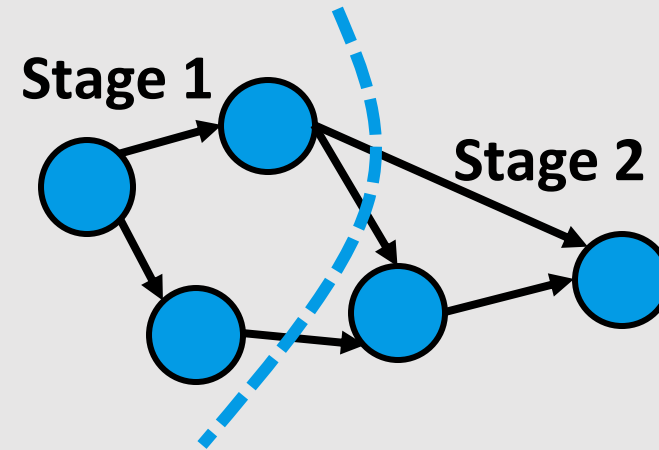


Iterative two-stage search algorithm

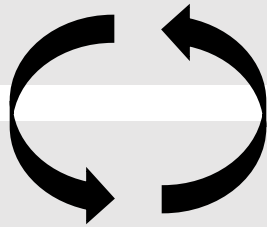
**Pipeline parallelism
dimension**



Algorithm: graph cut



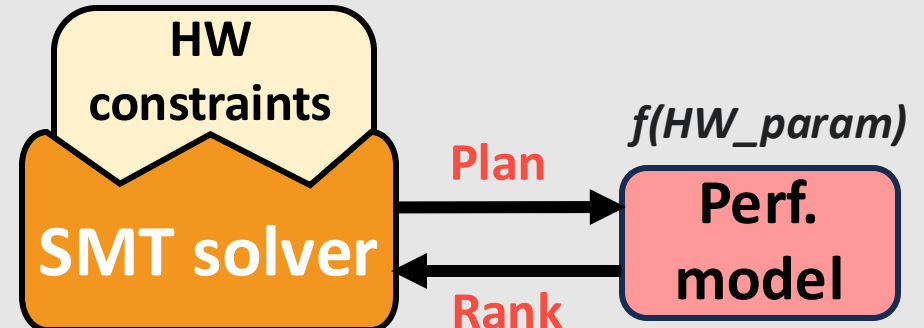
**Packet/flow
parallelism dimension**



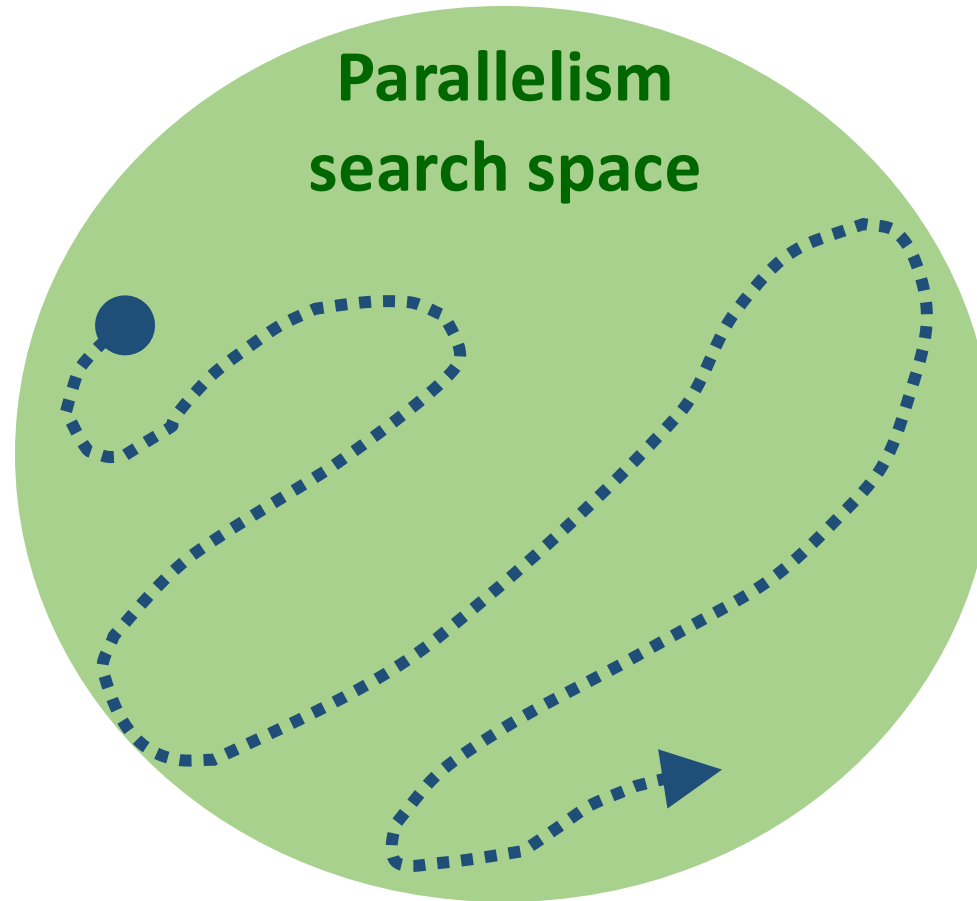
Algorithm: constraint satisfaction



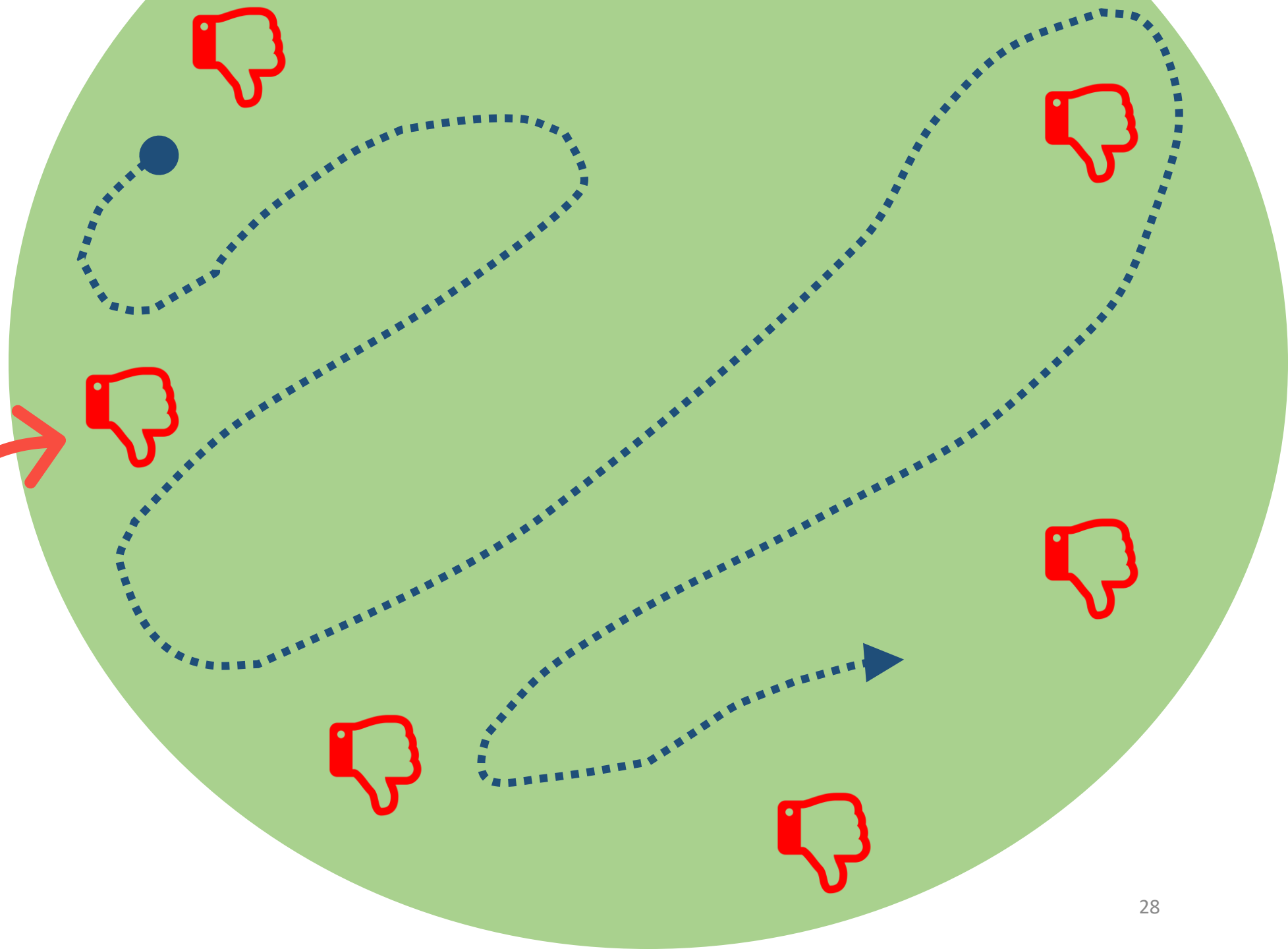
*(Core_Num,
Mem_Size, ...)*



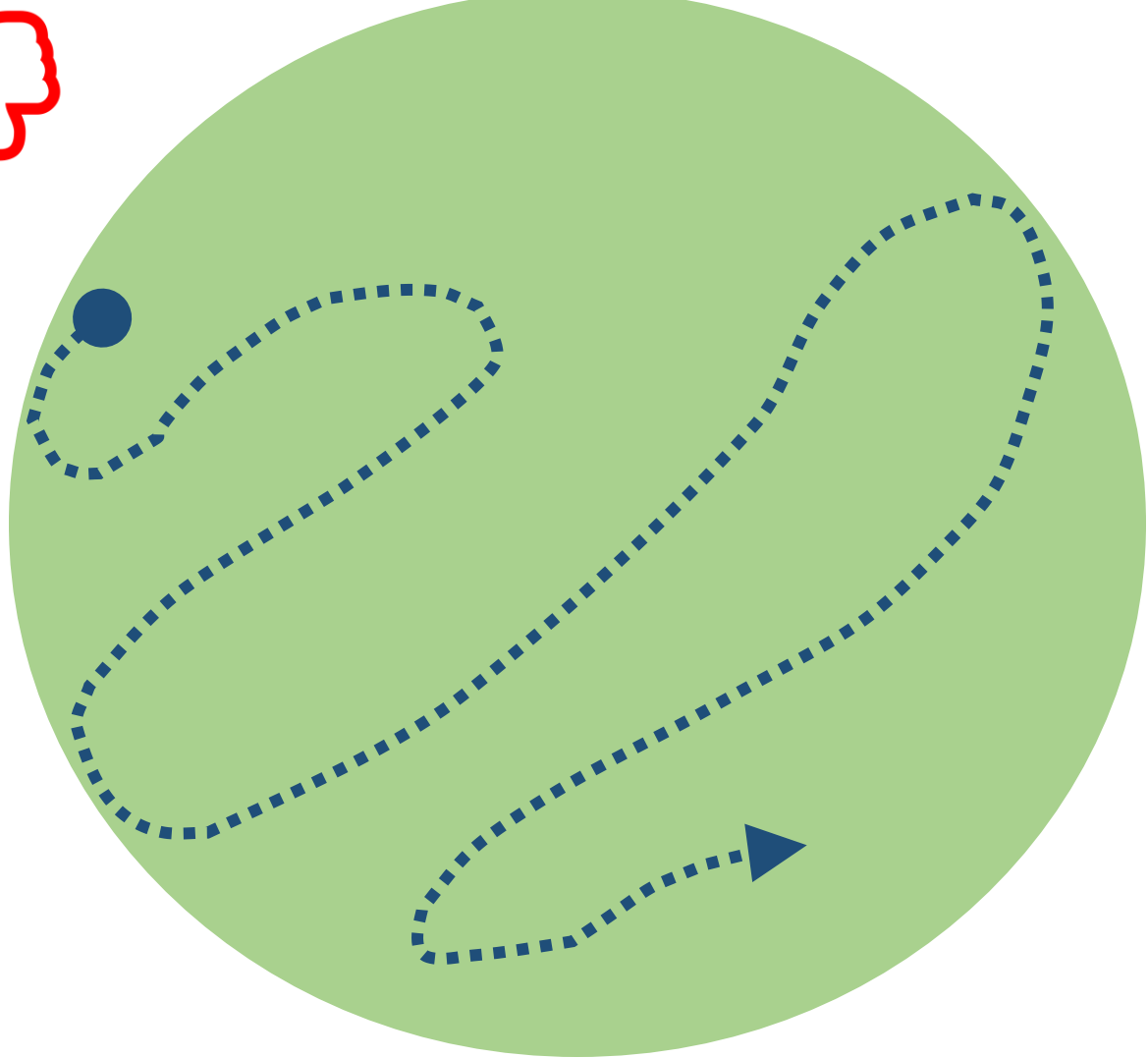
**Search
algorithm**



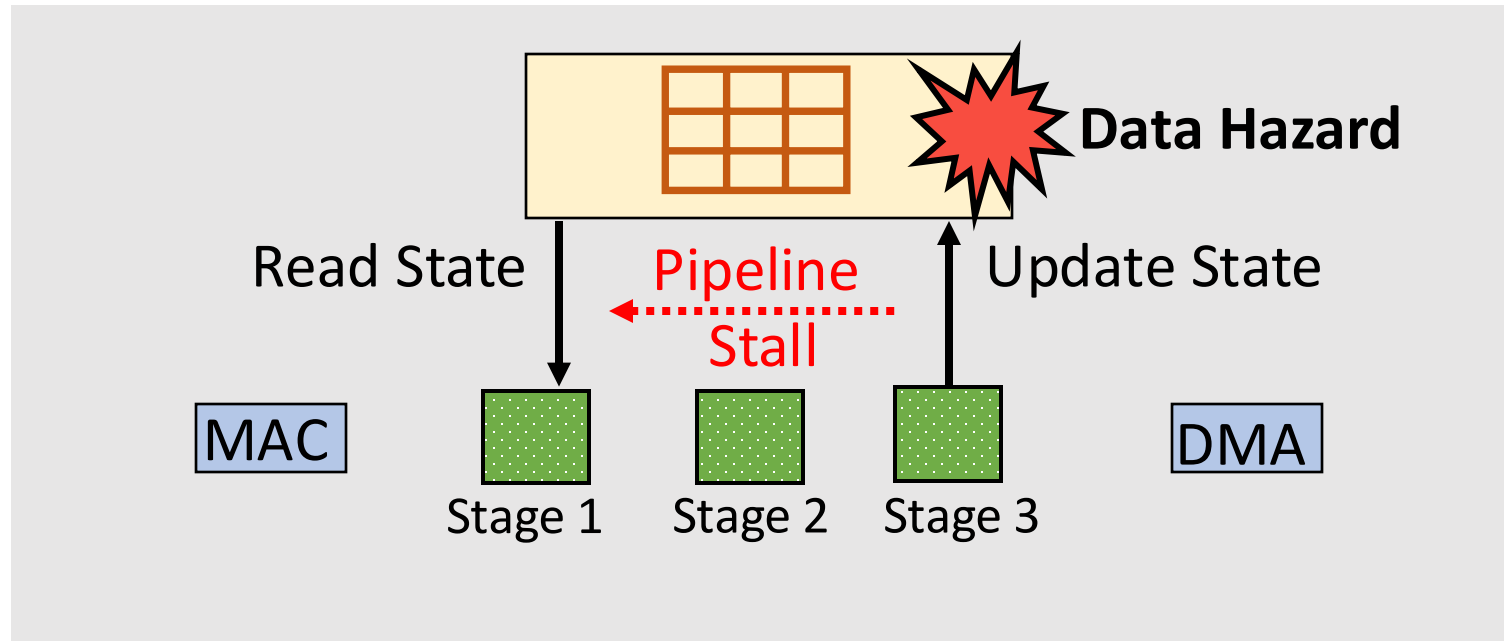
Bad plans



Bad plans

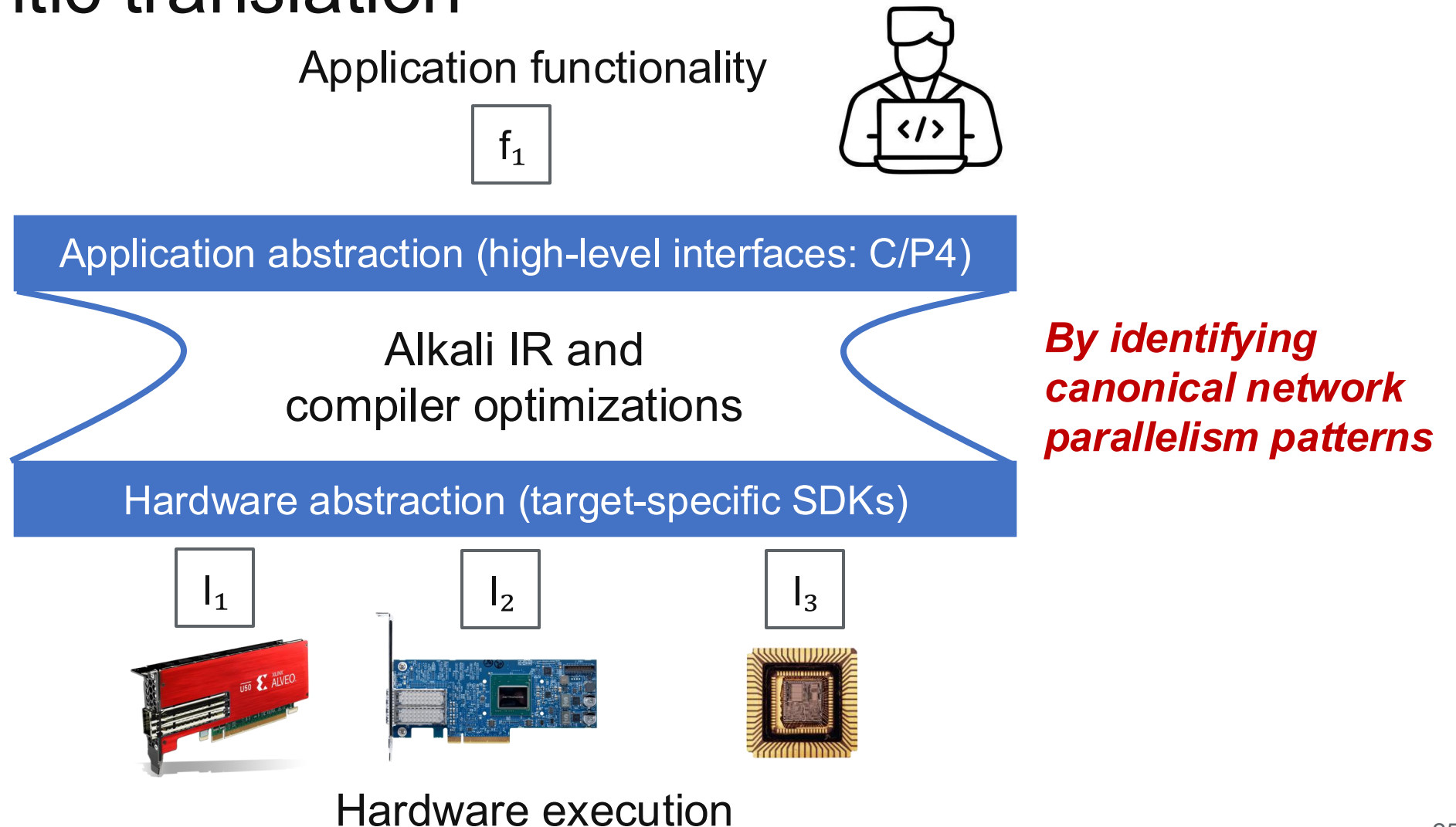


Example of a Bad Pipeline Plan



Prune this bad plan: pipeline cut should avoid splitting state

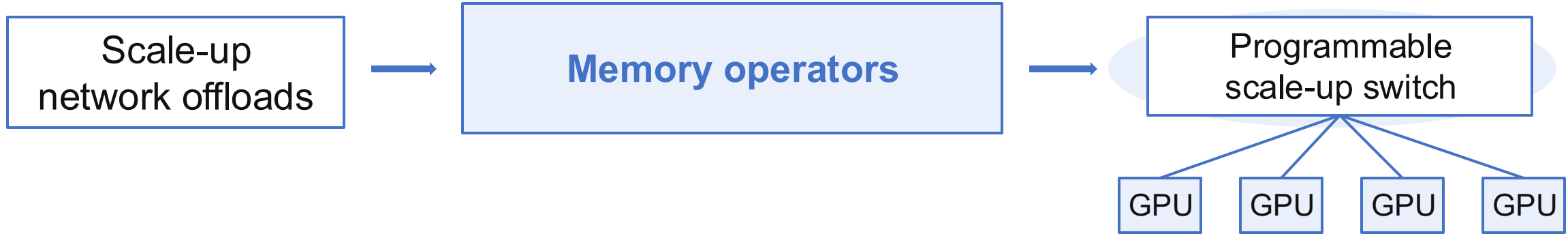
Finding the right abstraction for semantic translation



MemOps

HotNets '26

Programmability for memory transactions

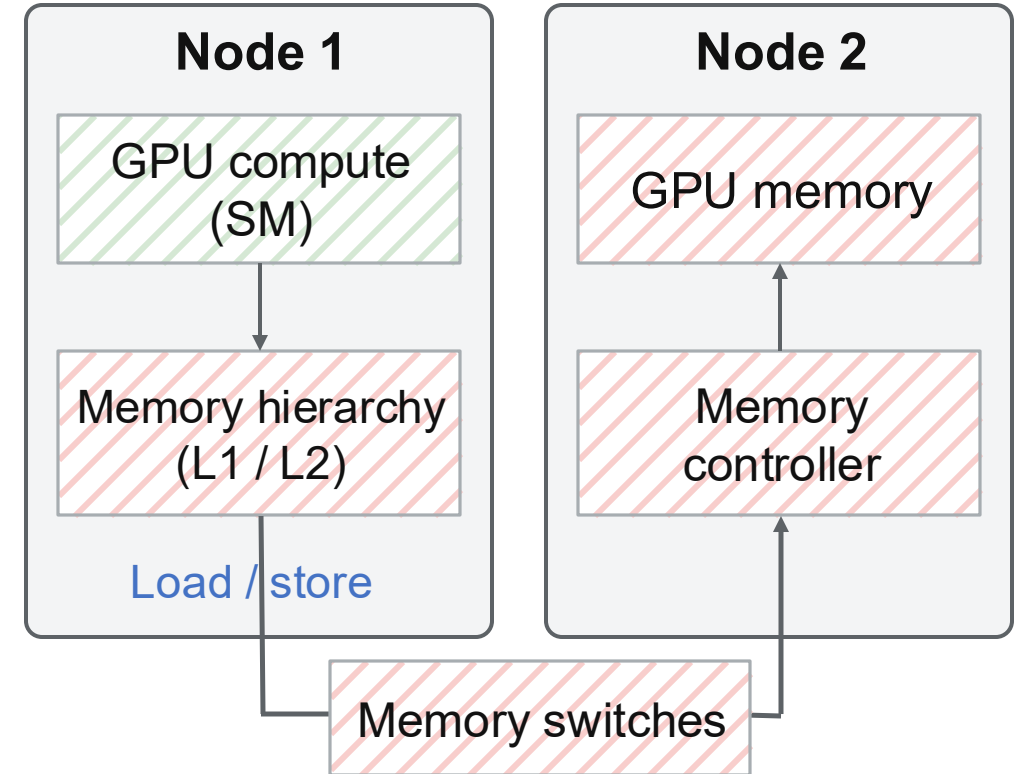
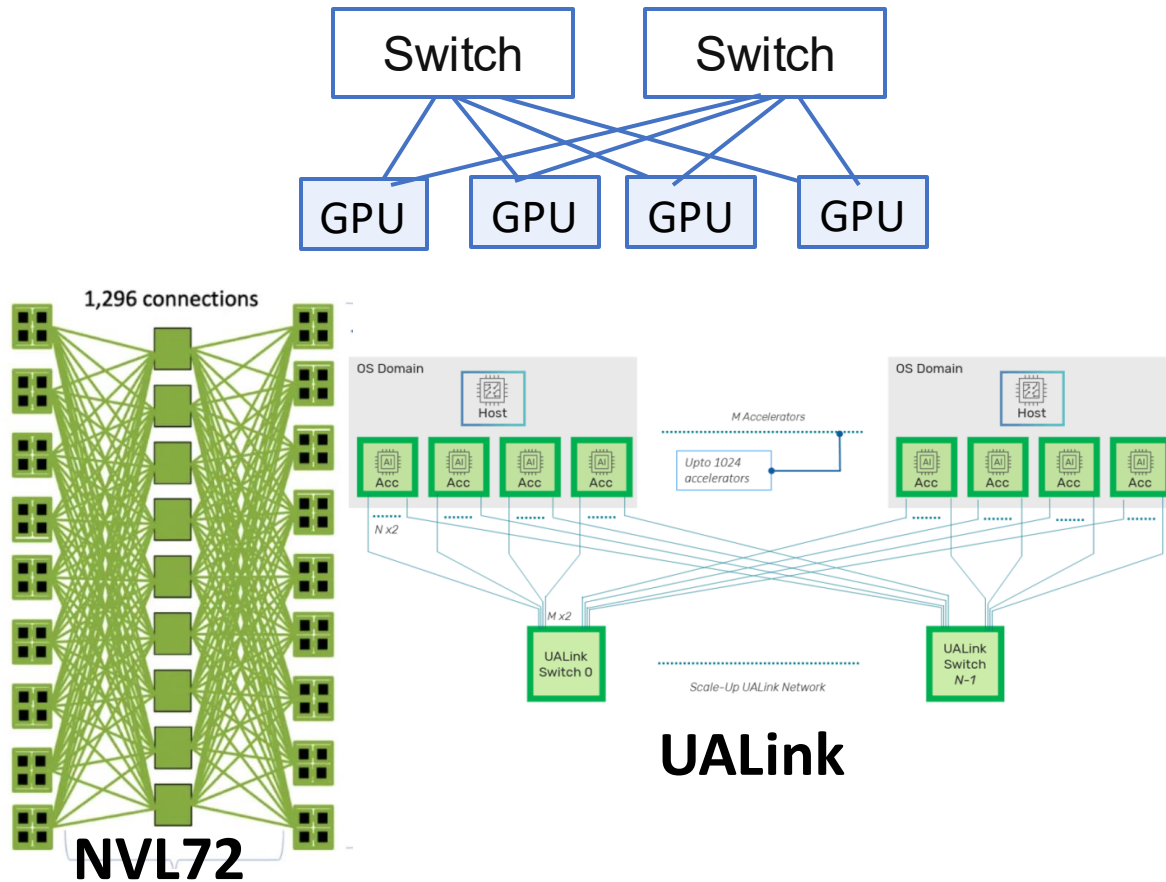


Programmable scale-up switches for software-defined GPU fabrics

Shouxu Lin, Ran Shu, Daehyeok Kim, Jiaxin Lin

HotNets 2026

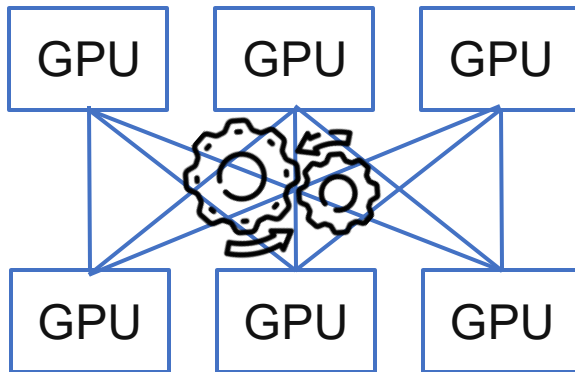
Today's scale-up data path is fixed-function



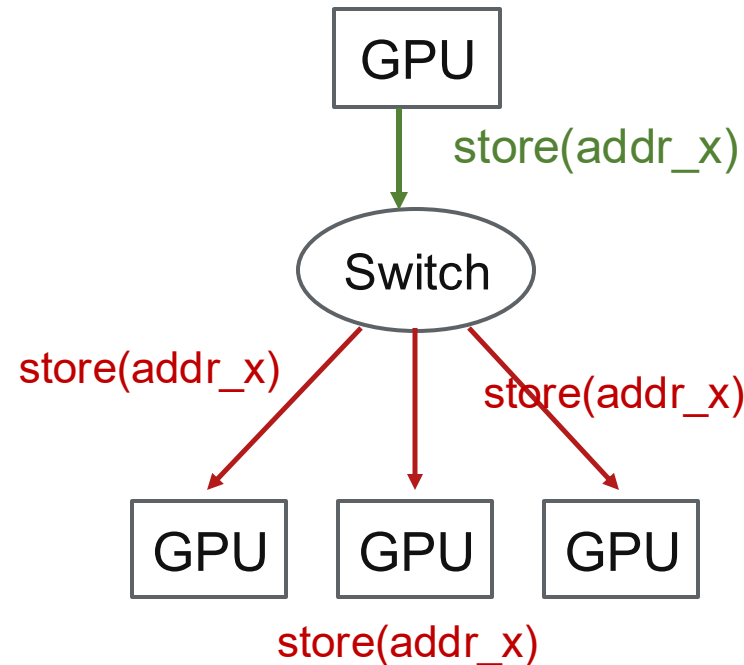
Green: programmable software or hardware
Red: fixed-function hardware

Why more programmability in scale-up fabrics?

Dynamic collective offload

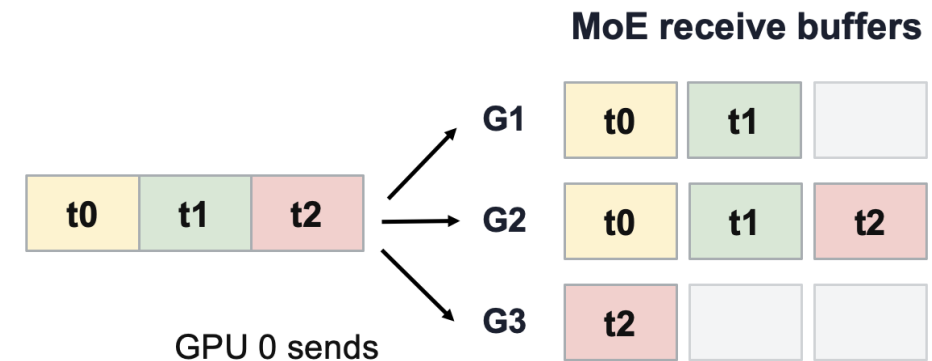


ML collective patterns are increasingly dynamic (e.g., MoE)



Existing switch offload:

- Preconfigured groups
- Symmetric offsets

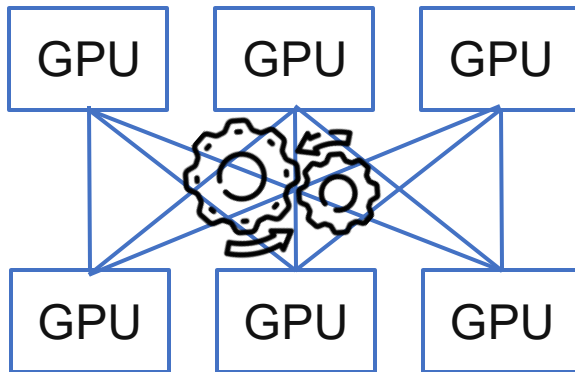


MoE workload requirements:

- Multicast destinations and memory addresses are determined at runtime

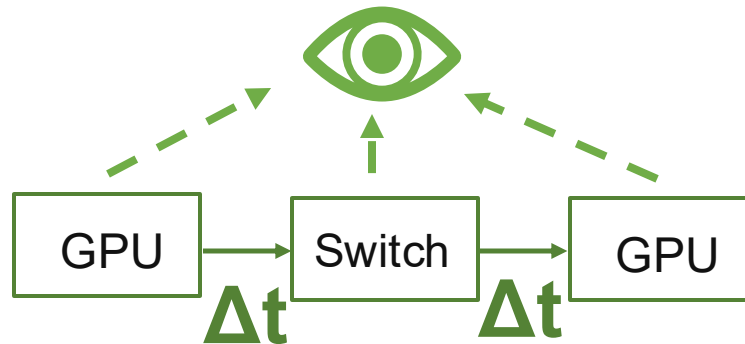
Why more programmability in scale-up fabrics?

Dynamic collective offload



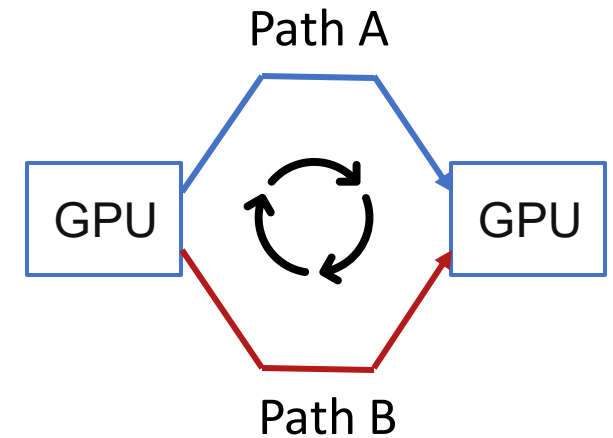
ML collective patterns are increasingly dynamic (e.g., MoE)

Fabric observability



Fabric latency breakdown and bottleneck diagnosis

Fabric control



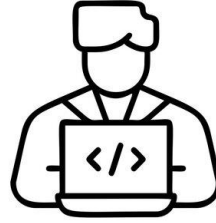
Adaptive paths under changing conditions

Vision: We should make scale-up fabrics software-defined, just as programmable switches have done for Ethernet over the past decade.

What is the right semantic abstraction for scale-up switches?

Application functionality

f_1



ML application abstraction? (E.g., collective, telemetry)

Scale-up switch compiler?

Scale-up switch hardware abstraction?

l_1



Hardware execution

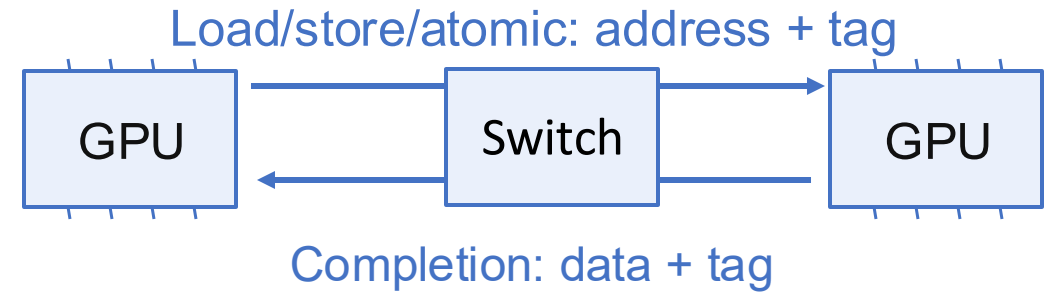
Requirements:

- ***Compatibility memory protocols***
- ***Expressive & bounded programmability***
- ***Verifiable correctness***

Computing in scale-up require memory semantic reasoning

- In-network computation will operate on distributed memory state
- It need to respect load/store/atomic ordering, visibility, and dependencies
- Expressed through memory addressing, not just 5-tuple flow metadata.
- Memory semantic may variance on GPU architecture and memory subsystem design.

Memory-semantic in scale-up

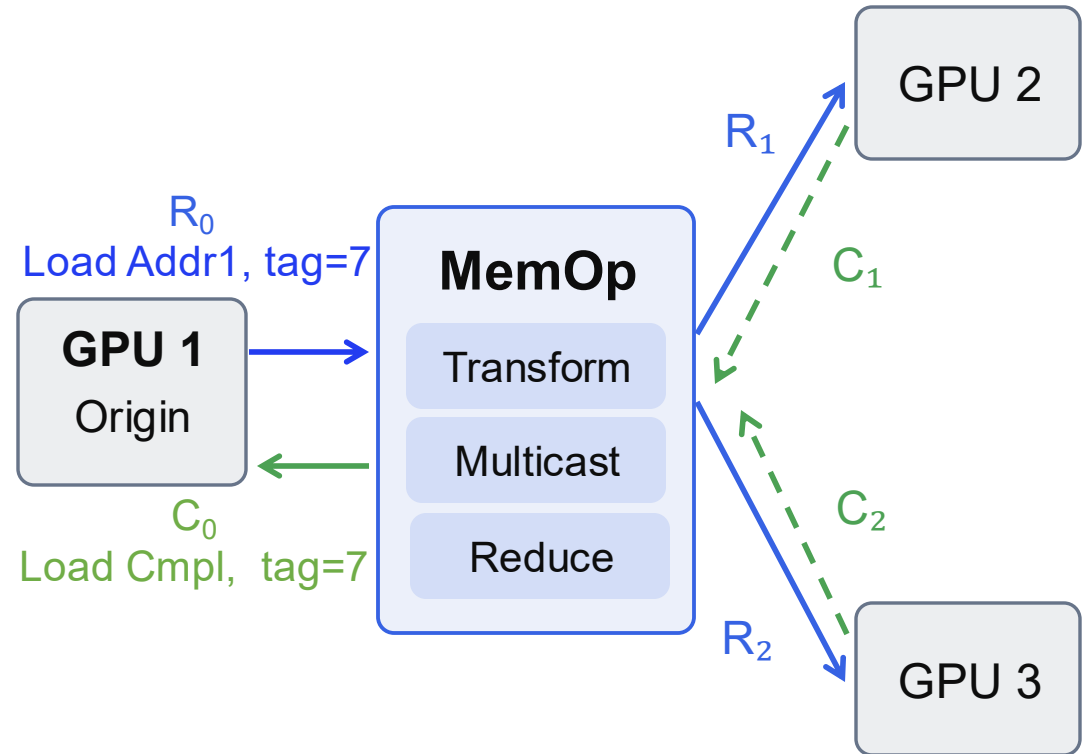


Semantic abstraction for memory switches

MemOps operate on fine-grained memory requests and completions

Each MemOp can

- Inspect/Transform request/completion
- Route or multicast requests
- Reduce multiple completions
- Compose MemOps into chains



Constraints:

- **Transaction guarantee:** every request must come with its matching completion.
- **Memory dependency:** req/cmpl must be processed/generated following certain order

Encoding constraints in programming interface

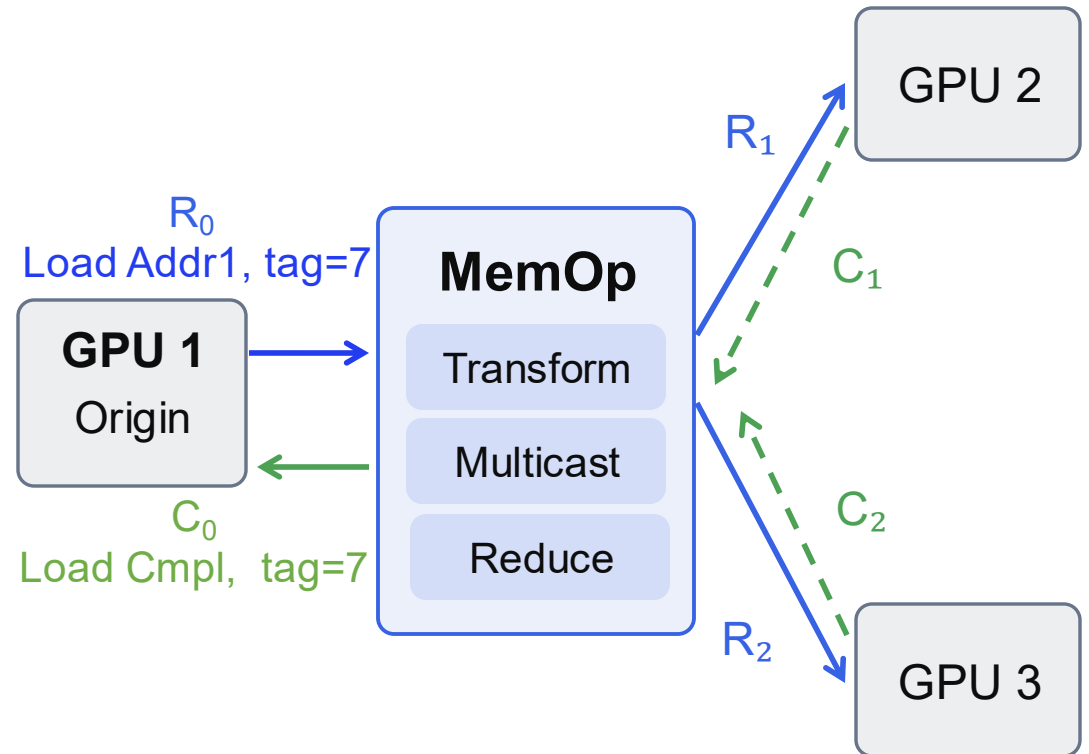
```
trigger(store_req req) {  
    // Fan out; retain parent dependencies  
    cmpls = req.multicast(addr1, addr2)  
        .send();  
  
    // Wait for matching child completions  
    result = await_all(cmpls);  
  
    // Reply to the original request  
    req.cmpl(result).send();  
}
```

Transaction Invariant:

Preserve the parent tag: $\text{tag}(\mathbf{C}) = \text{tag}(\mathbf{R})$.

Memory Dependency:

- Reply to R only after **both $\mathbf{C}_1$ and $\mathbf{C}_2$** arrive.

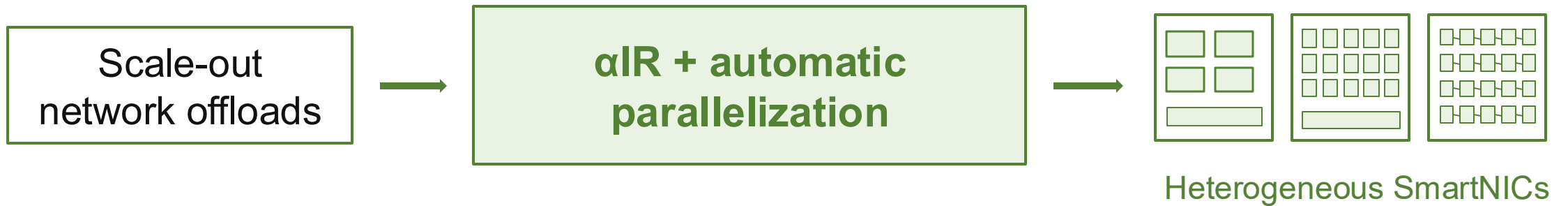


Programmability across heterogeneous networks

Alkali

NSDI '25

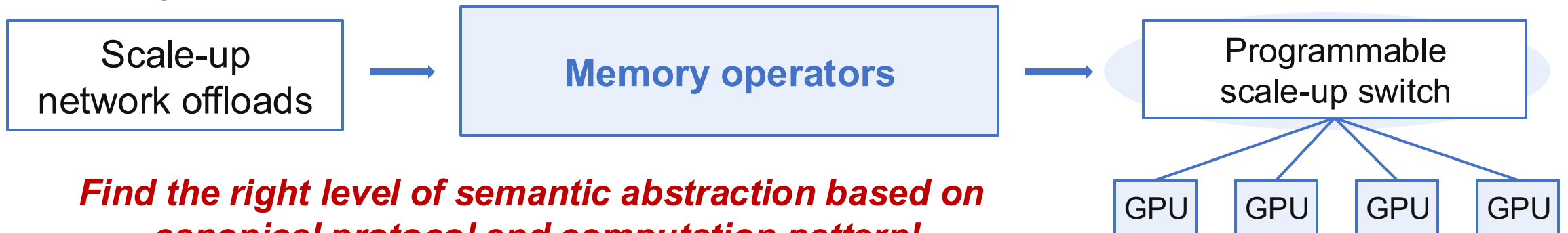
Portable compiler across SmartNICs



MemOps

HotNets '26

Programmability for memory transactions



Find the right level of semantic abstraction based on canonical protocol and computation pattern!