Enabling Portable and High-Performance SmartNIC Programs with Alkali

Jiaxin Lin*[1], Zhiyuan Guo*[2], Mihir Shah[1], Tao Ji[1], Yiying Zhang[2], Daehyeok Kim[1] and Aditya Akella[1]

1 TEXAS The University of Texas at Austin

2 UC San Diego

SmartNIC Trends

RSS

Middlebox

Transport

IPSec

LSO

Msg Process

Net Function

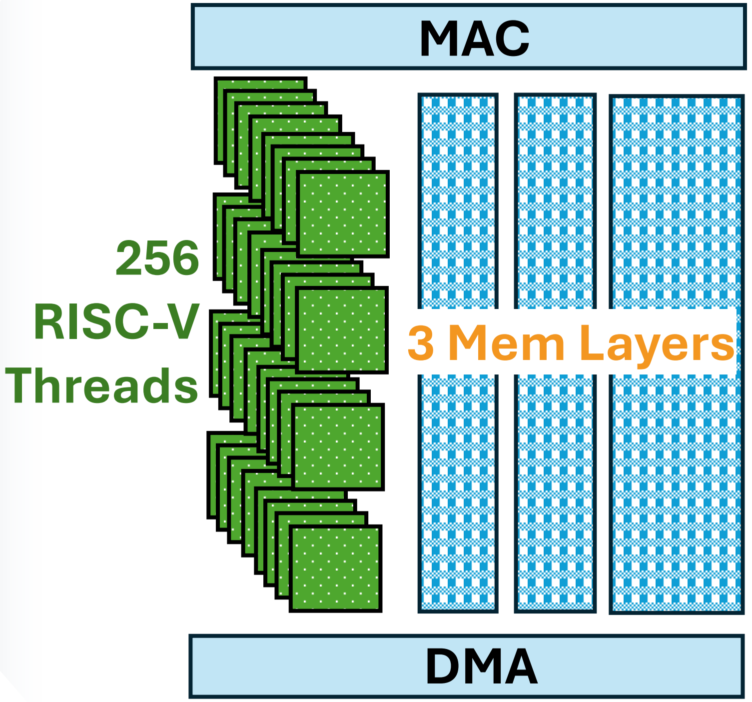
Trend 1:Increasing number of applications

Intensifies programming barriers

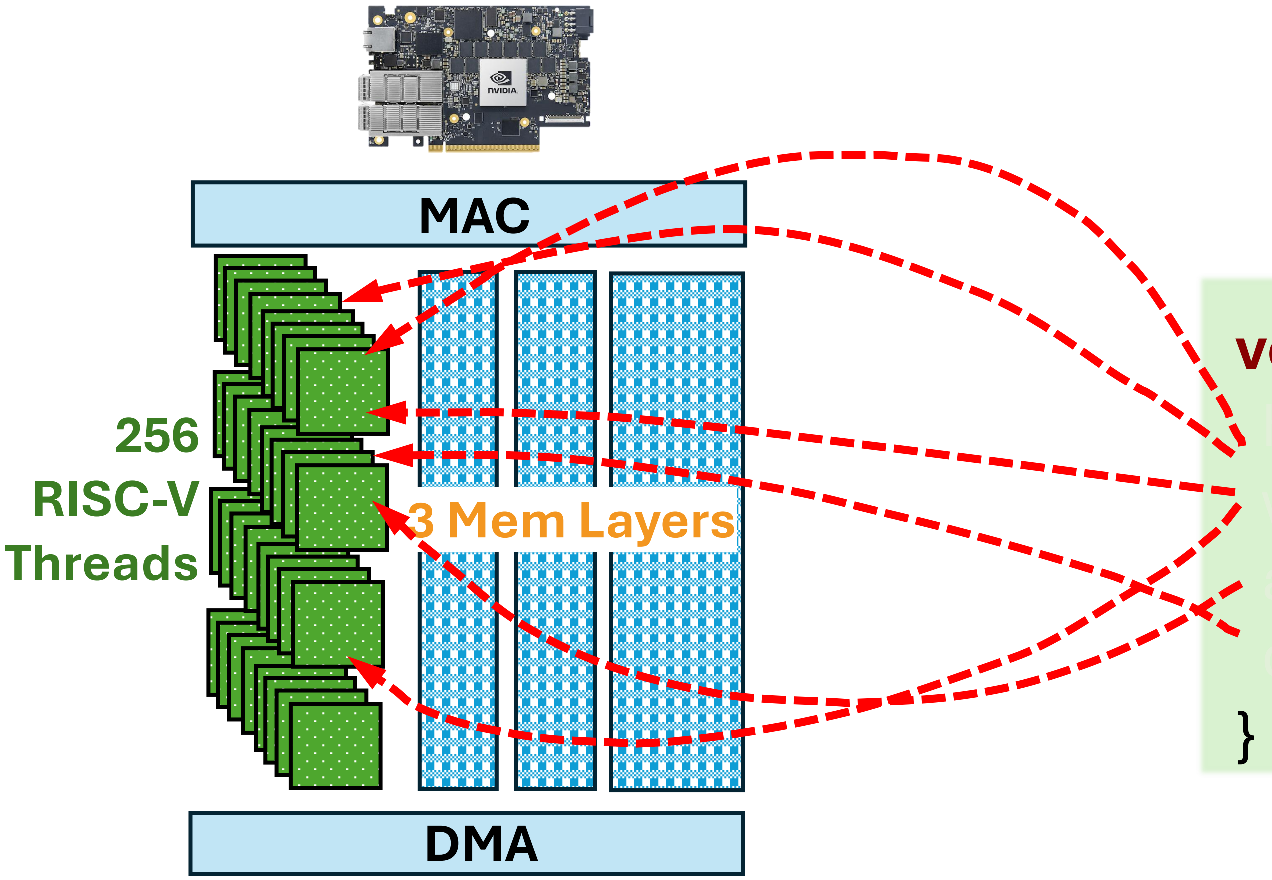
Trend 2: Increasing number of hardware variants

SmartNIC Rich Hardware Parallelism

NVIDIA
BlueField-3 DPA



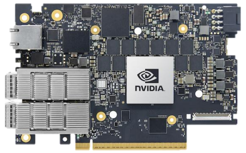
Barriers of SmartNIC Programming
#1 Low level parallel programming

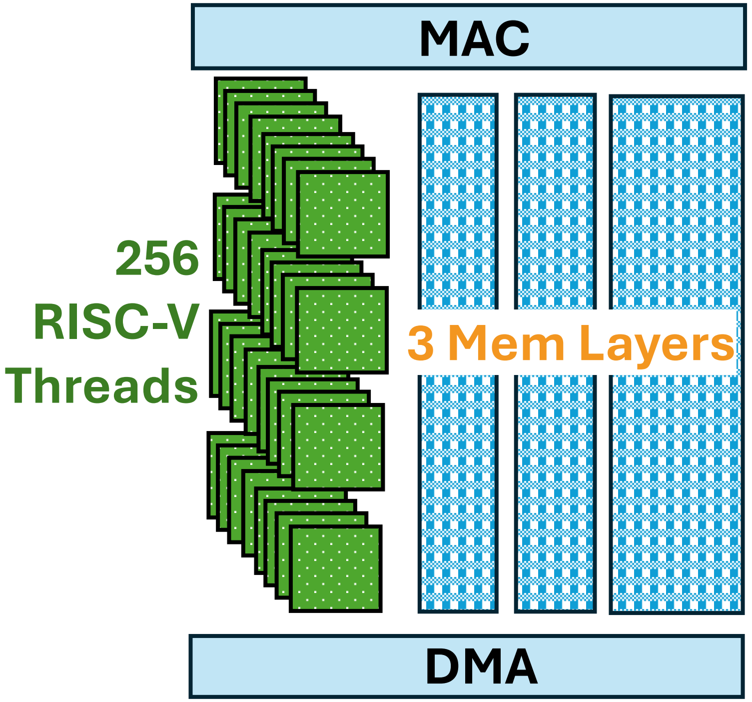



```
void TCP_offload(pkt e){
  hdr = parse(e);
  win_update(hdr, win);
  ack_gen(hdr);
  dma_write(e);
}
```

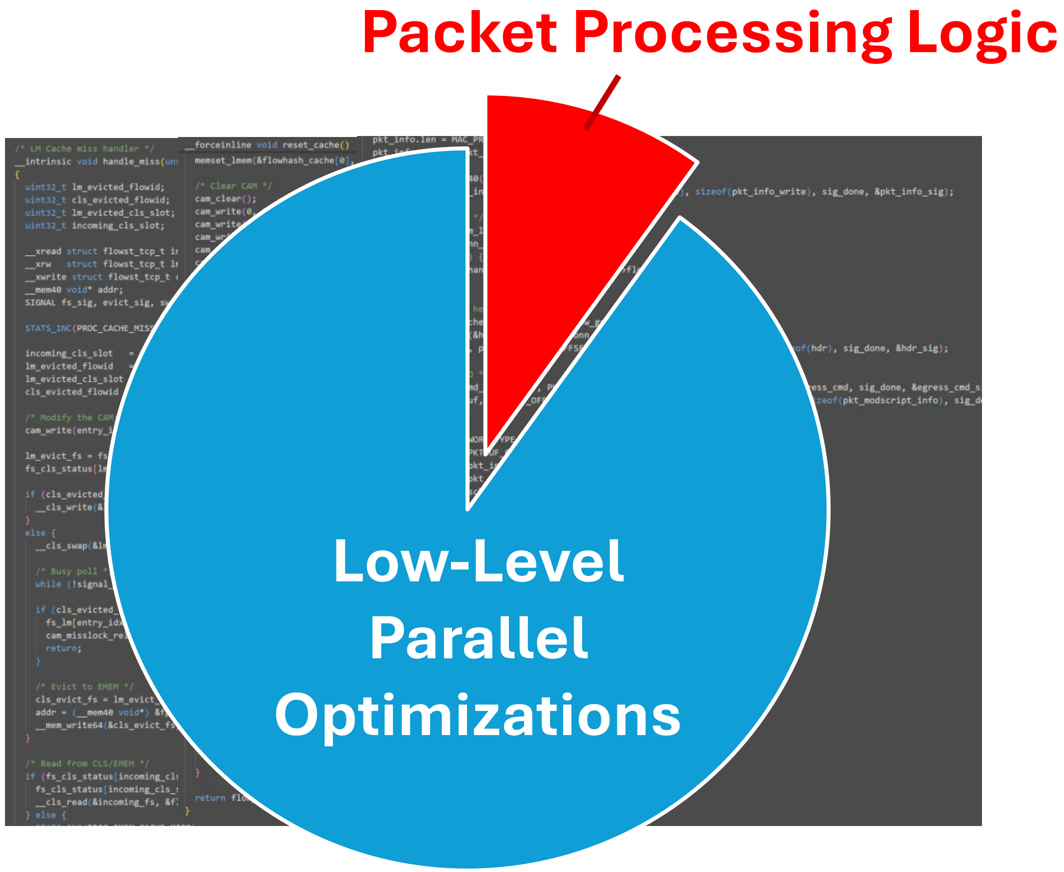
Low-level interfaces:
DOCA with **500+** MicroC lib functions

Barriers of SmartNIC Programming
#1 Low level parallel programming





Low-level interfaces:
DOCA with **500+** MicroC lib functions



Lines of Code Distribution

Barriers of SmartNIC Programming
#2 Non-portability

Agilio

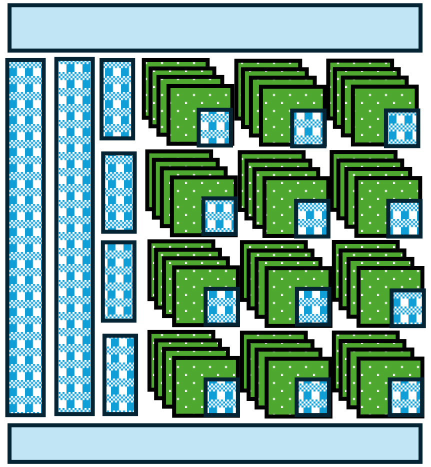
FPGA

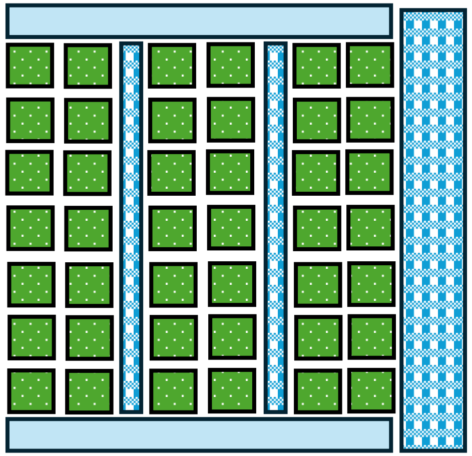
BlueField-2

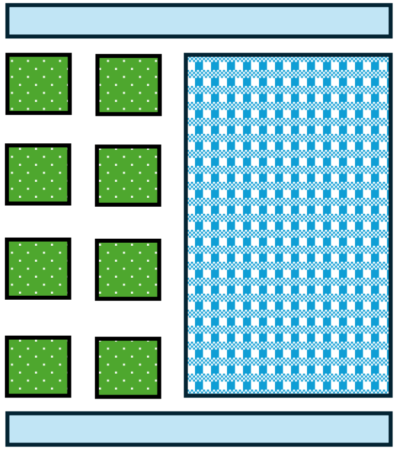
Interfaces:

MicroC with Macros	Verilog (RTL)	DPDK + extern libc

Architectures:

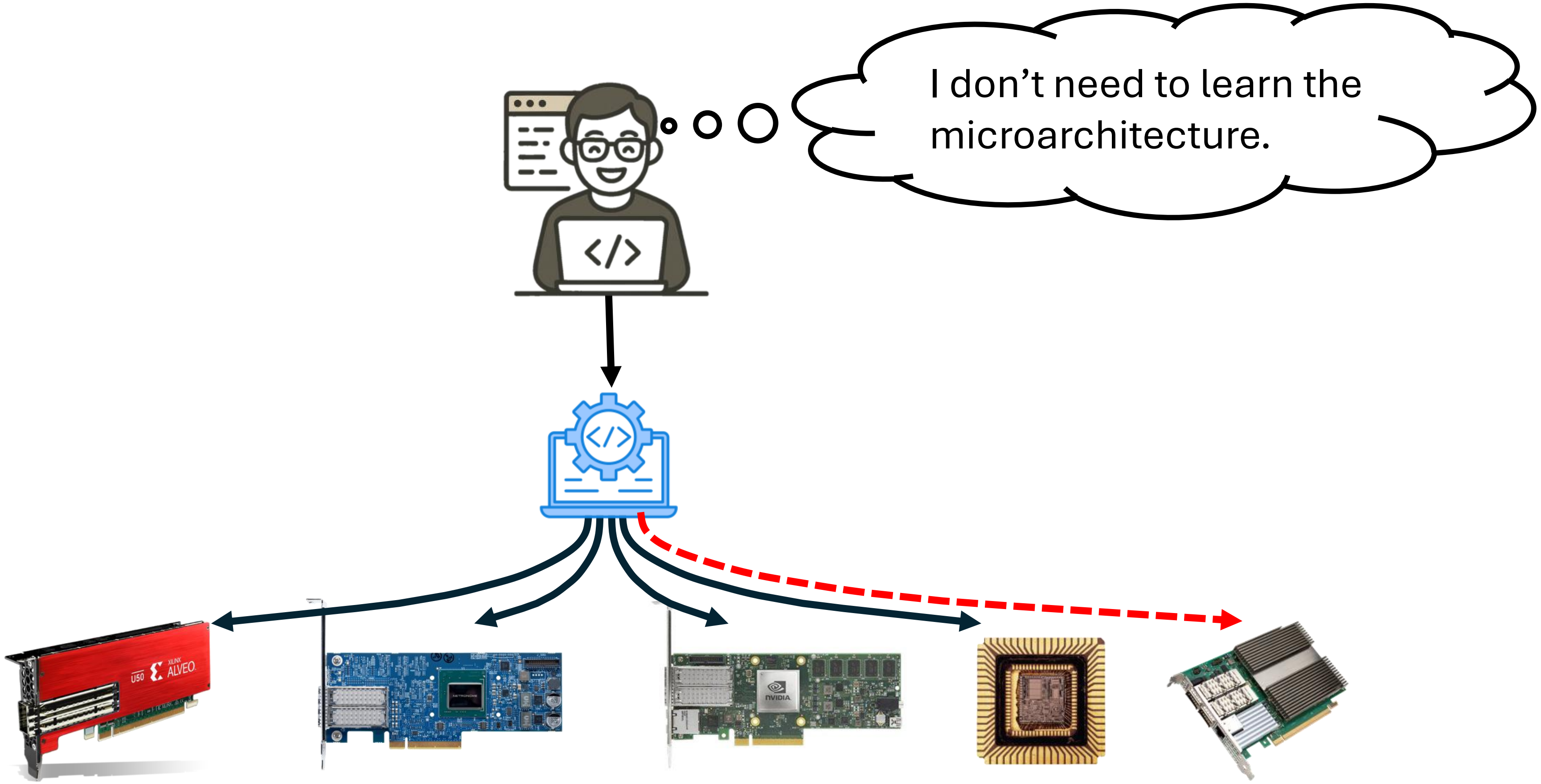






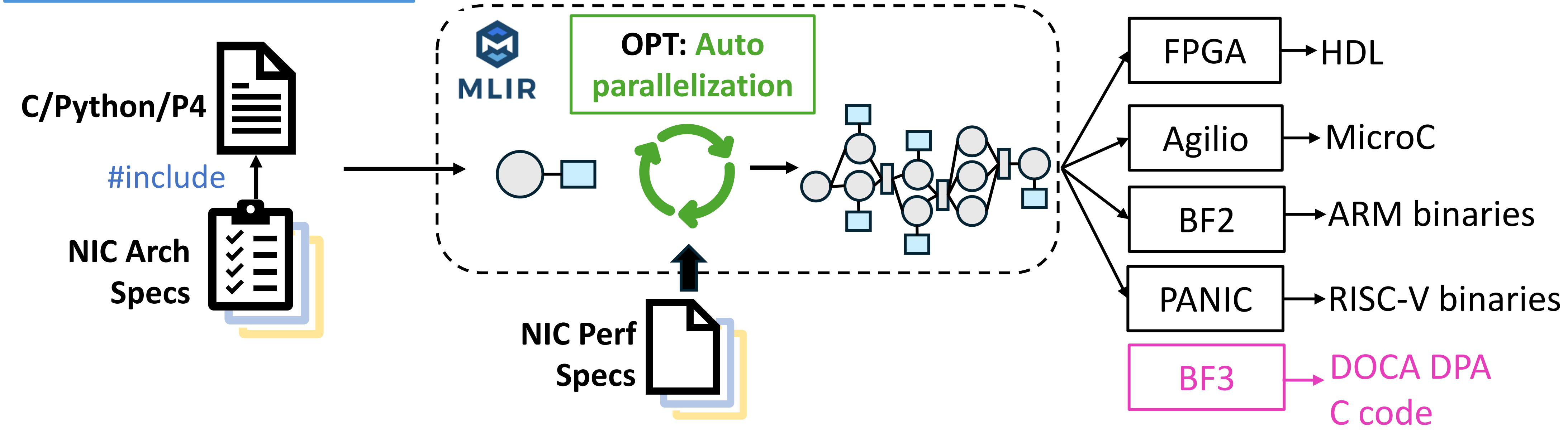
An Ideal Programming Framework

Target-agnostic Programming	Portability & Performance	Extensible



Alkali: A Multi-Target Compilation Framework for NICs

Target-agnostic Programming ✓	Portability & Performance ✓	Extensible ✓
Programming: Single threaded event processing	**Alkali Compiler:** Event handler graph αIR	**NIC Backends:** Simple code gen



Talk Outline

- Target agonistic programming interfaces
- Event handler graph-based αIR
- Auto parallelization optimization
- Demo and future plan

Target Agonistic Programming Interfaces

- **Run-to-completion**, single threaded program.
  - Process and generate **hardware events**.
- Import architecture specification, defines supported events.
  - **Portable if:** two NICs have the same arch spec.

```
# include<agilio_spec.h>
void _net_recv(pkt e){
  mac_hdr mac = buf_extract(e, 48);
  tb_update(table1, mac, 1);
  _dma_write(e, 0x8000);
}
```

agilio_spec.h

```
void _net_recv(hdr_t hdr,buf_t data){}
void _dma_write(...){}
void _dma_read(...){}
void _mmio_doorbell(...){}
```

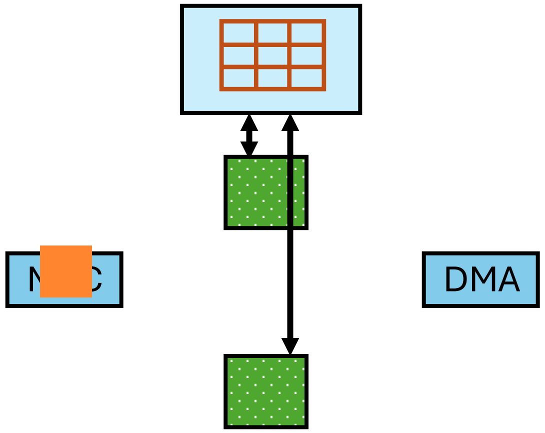
Talk Outline

- Target agonistic programming interfaces
- **Event handler-graph-based αIR**
- Auto parallelization optimization
- Demo and future plan

αIR Design

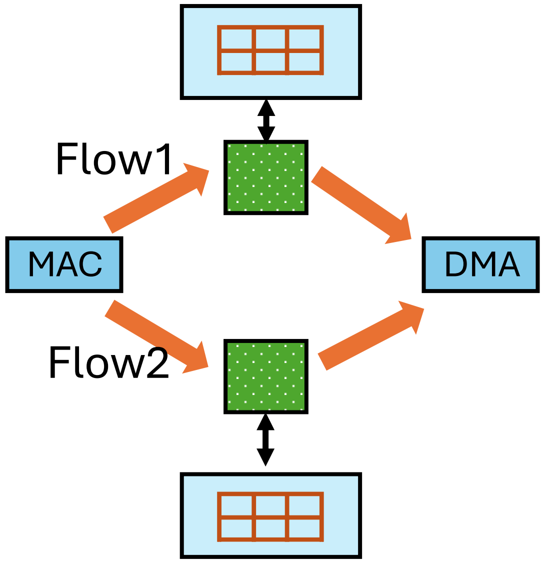
- A common representation captures parallel execution patterns on NICs.

Type 1: Packet Parallelism



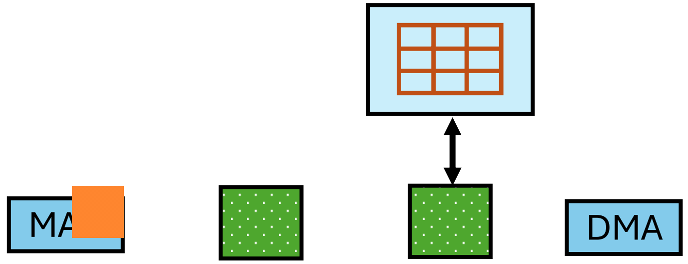
Pros: Maximizes parallelism.
Cons: Requires synchronization for state.

Type 2: Flow Parallelism



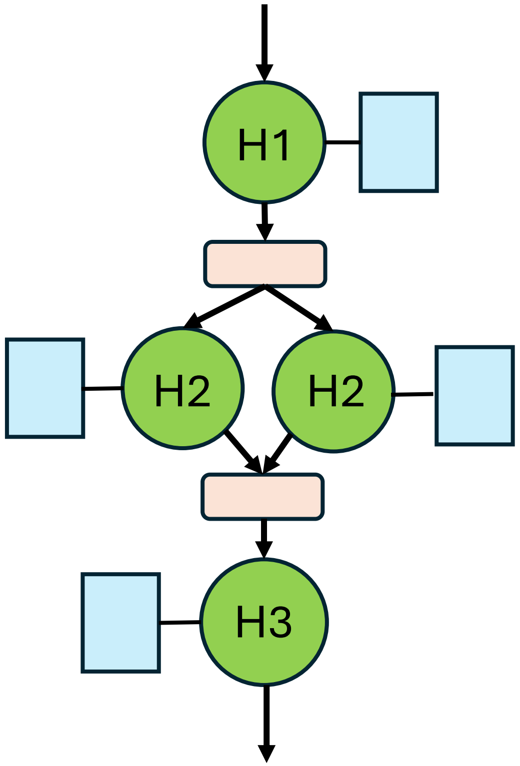
Pros: No state synchronization.
Cons: Does not support global state

Type 3: Pipeline Parallelism



Pros: Supports global state.
Cons: Communication overhead.

Express Three Parallelisms: Event Handler Graph



Event Handler:

- Code block in a compute unit.
- Can be replicated.

Events:

- Triggers handler's computation.

Event Controller:

- Defines event steering and ordering rules among handler and its replicas.

Persistent State:

- State that persists across events, e.g., flow table, counters.

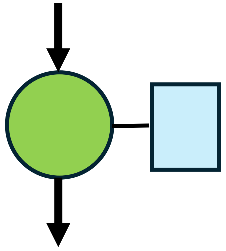
```
module {
  ep2.func private @__handler_NET_SEND_main_send(%arg0: !ep2.context, %arg1: !ep2.buf) attributes {atom = "main_send", event = "NET_SEND", extern =
    "ep2.terminate"() : () -> ()
  }
  ep2.func private @__handler_NET_RECV_main_recv(%arg0: !ep2.context, %arg1: !ep2.buf) attributes {atom = "main_recv", event = "NET_RECV", type = "
    %0 = "ep2.init"() : () -> !ep2.struct<"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16>
    %1 = "ep2.init"() : () -> i48
    %2 = "ep2.init"() : () -> !ep2.buf
    %3 = "ep2.extract"(%arg1) : (!ep2.buf) -> !ep2.struct<"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16>
    %4 = ep2.struct_access %3[1] : <"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16> -> i48
    %5 = ep2.struct_access %3[1] : <"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16> -> i48
    %6 = ep2.struct_access %3[0] : <"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16> -> i48
    %7 = "ep2.struct_update"(%3, %6) <{index = 1 : i64}> : (!ep2.struct<"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16>, i48) -> !e
    %8 = ep2.struct_access %7[0] : <"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16> -> i48
    %9 = "ep2.struct_update"(%7, %4) <{index = 0 : i64}> : (!ep2.struct<"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16>, i48) -> !e
    "ep2.emit"(%2, %9) : (!ep2.buf, !ep2.struct<"eth_header_t" : isEvent = false, elementTypes = i48, i48, i16>) -> ()
    %10 = "ep2.nop"() : () -> none
    "ep2.emit"(%2, %arg1) : (!ep2.buf, !ep2.buf) -> ()
    %11 = "ep2.nop"() : () -> none
    %12 = "ep2.constant"() <{value = "main_send"}> : () -> !ep2.atom
    %13 = "ep2.init"(%12, %arg0, %2) : (!ep2.atom, !ep2.context, !ep2.buf) -> !ep2.struct<"NET_SEND" : isEvent = true, elementTypes = !ep2.context,
    ep2.return %13 : !ep2.struct<"NET_SEND" : isEvent = true, elementTypes = !ep2.context, !ep2.buf>
    "ep2.terminate"() : () -> ()
```

Expressed as a Dialect in MLIR

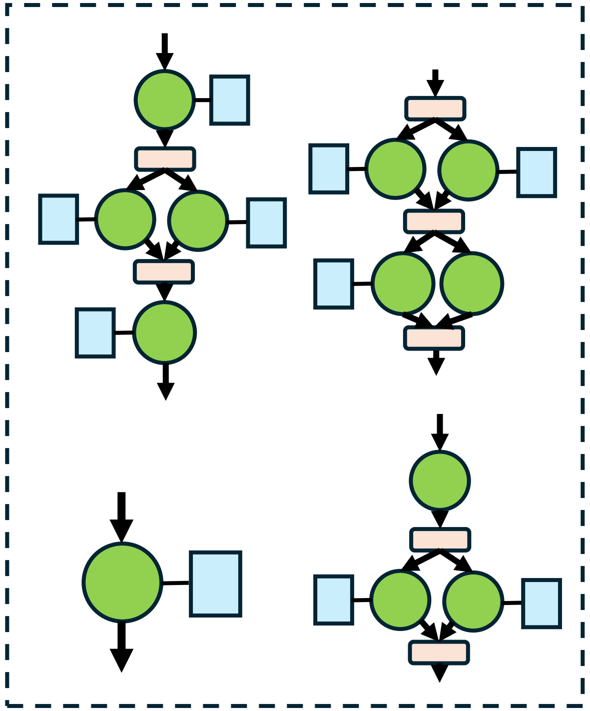
Talk Outline

- Target agonistic programming interfaces
- Event handler-graph-based αIR
- **Auto parallelization optimization**
- Demo and future plan

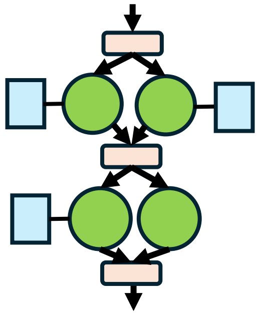
Auto Parallelization Optimization



Single Threaded Input



Search all semantic equivalent IR graphs



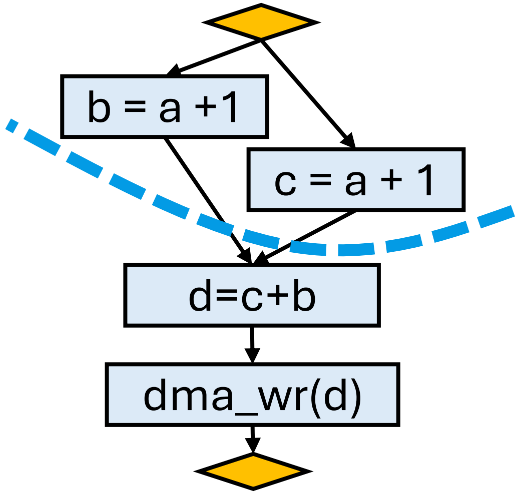
Best IR graph that runs fastest on the target NICs

Iterative Two-stage Algorithm to Guide the Search

Pipeline Cutting Engines

Algorithm: Graph Cut

handler (int a){
b = a+1;
c = a+1;
d = c+b;
dma_wr(d);
}



handler_1(int a){
b = a+1;
c = a+1;
gen_event(c, b);}

handler_2(int c, b){
d = c+b;
dma_wr(d);}

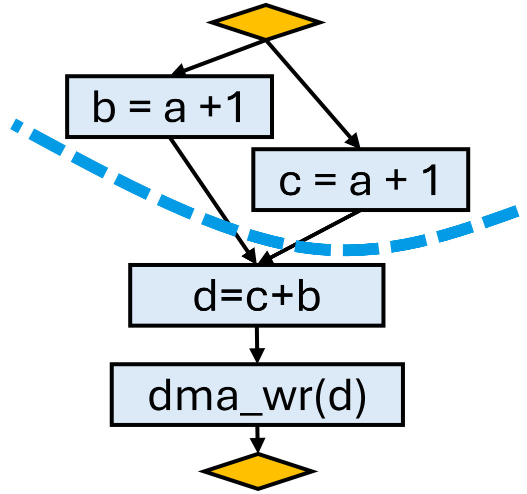
Mapping Engine

Iterative Two-stage Algorithm to Guide the Search

Pipeline Cutting Engines

Algorithm: Graph Cut

```
handler (int a){
 b = a+1;
 c = a+1;
 d = c+b;
 dma_wr(d);
}
```



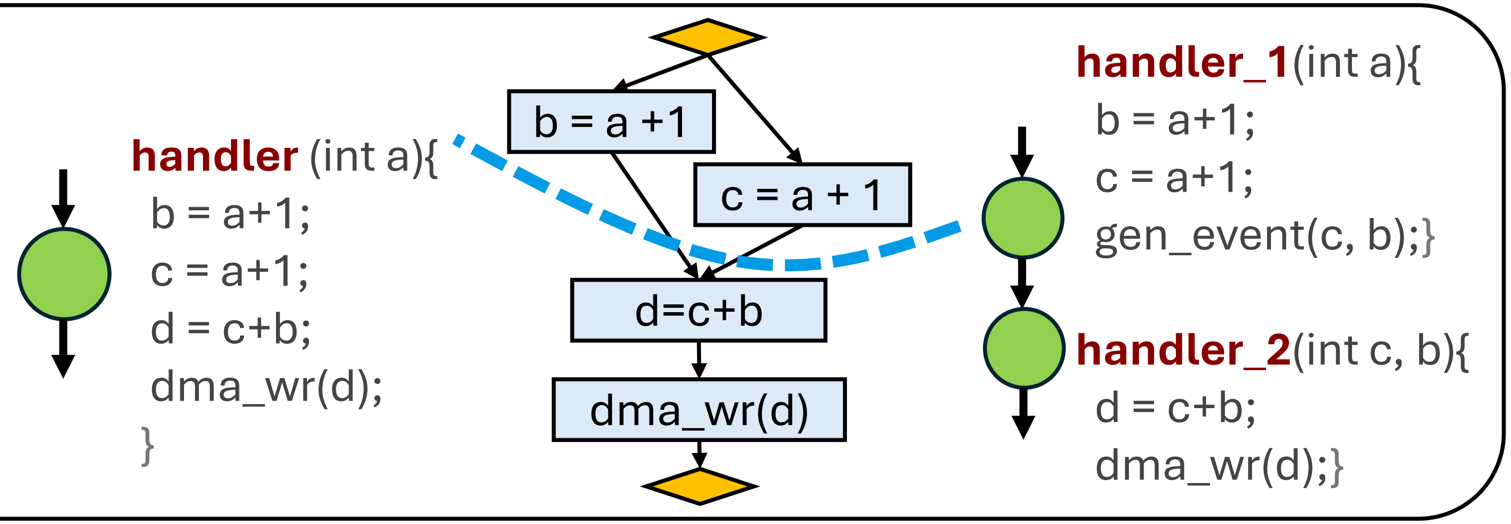

```
handler_1(int a){
 b = a+1;
 c = a+1;
 gen_event(c, b);}

handler_2(int c, b){
 d = c+b;
 dma_wr(d);}
```

Mapping Engine

Iterative Two-stage Algorithm to Guide the Search

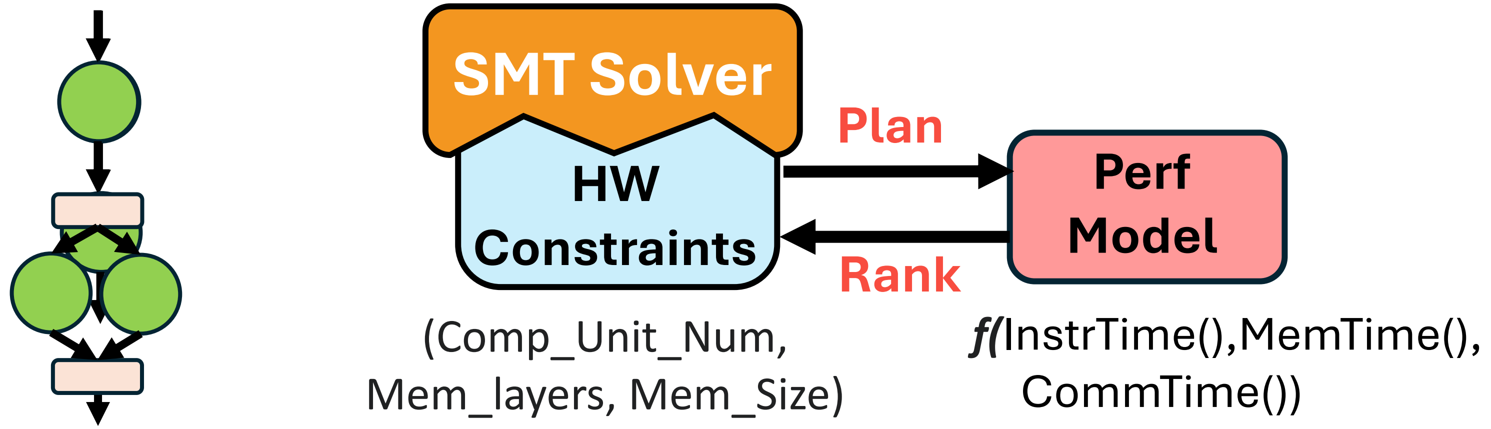
Pipeline Cutting Engines

Algorithm: Graph Cut



Mapping Engine

Algorithm: Constraint Satisfaction Problem



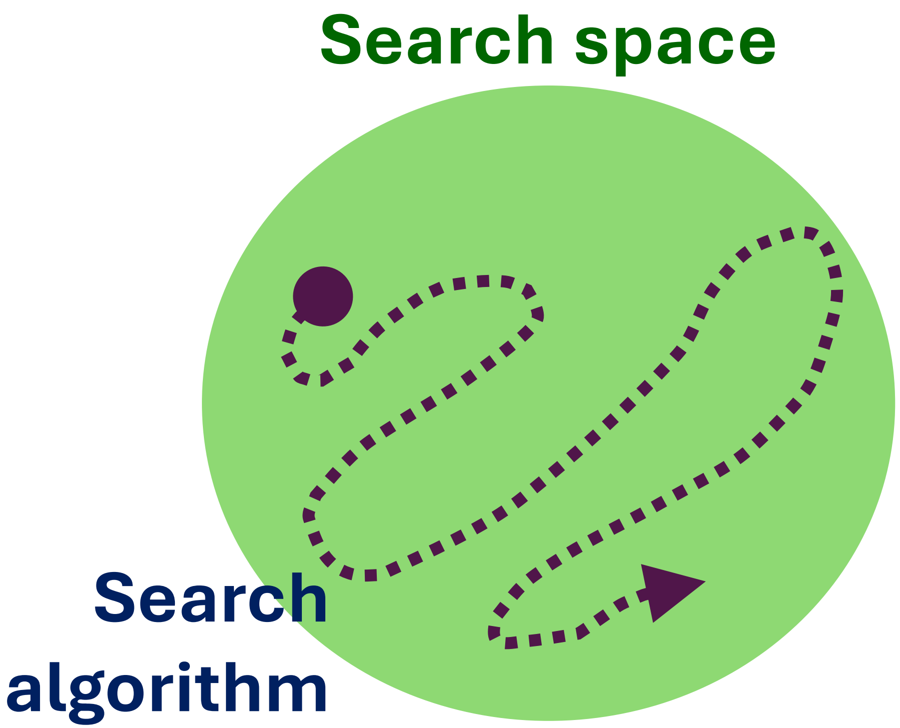
Adding Constraints to Improve Efficiency

Pipeline Cutting Engines

Algorithm: Graph Cut

Mapping Engine

Algorithm: Constraint Satisfaction Problem



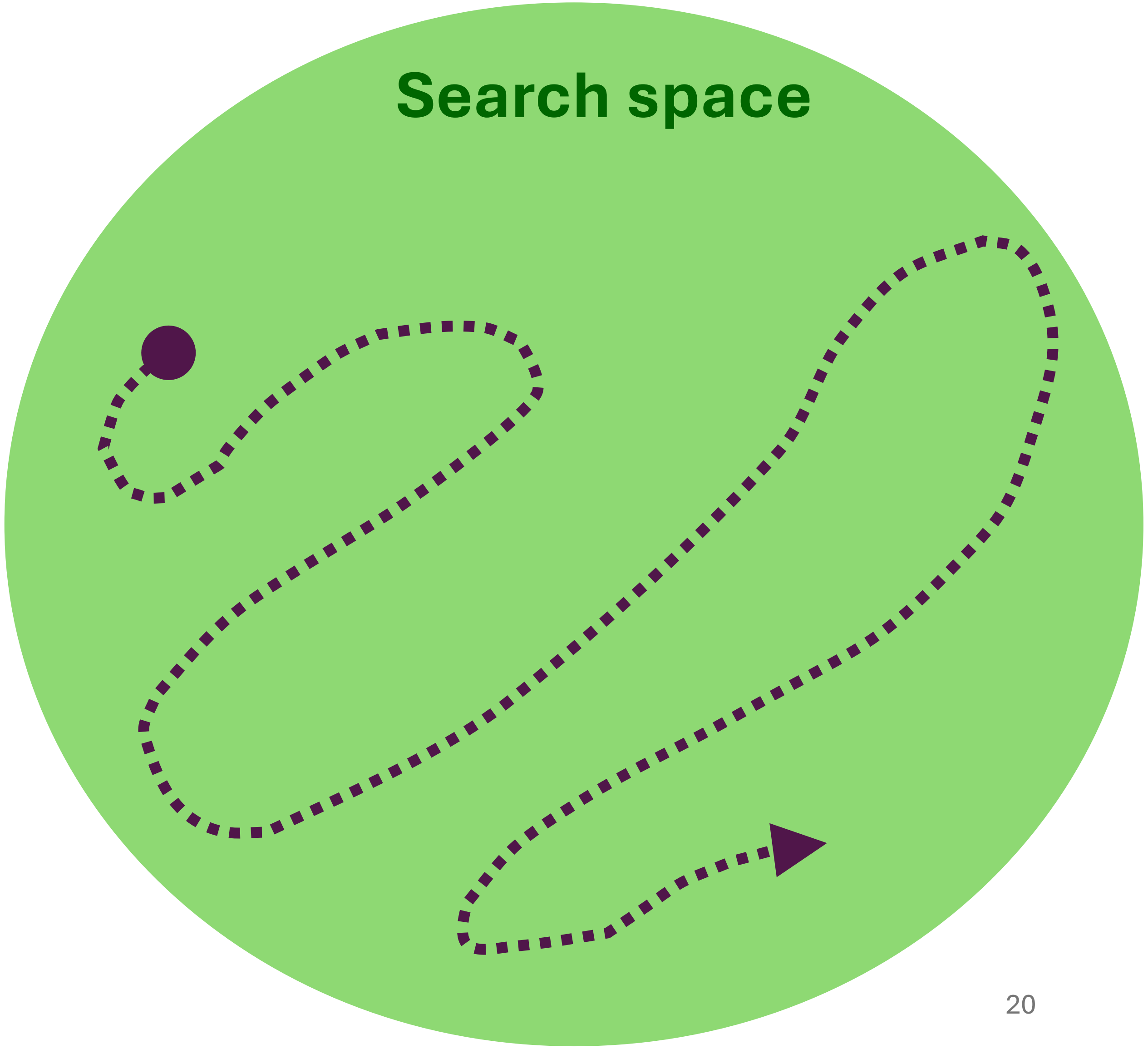
Adding Constraints to Improve Efficiency

Pipeline Cutting Engines

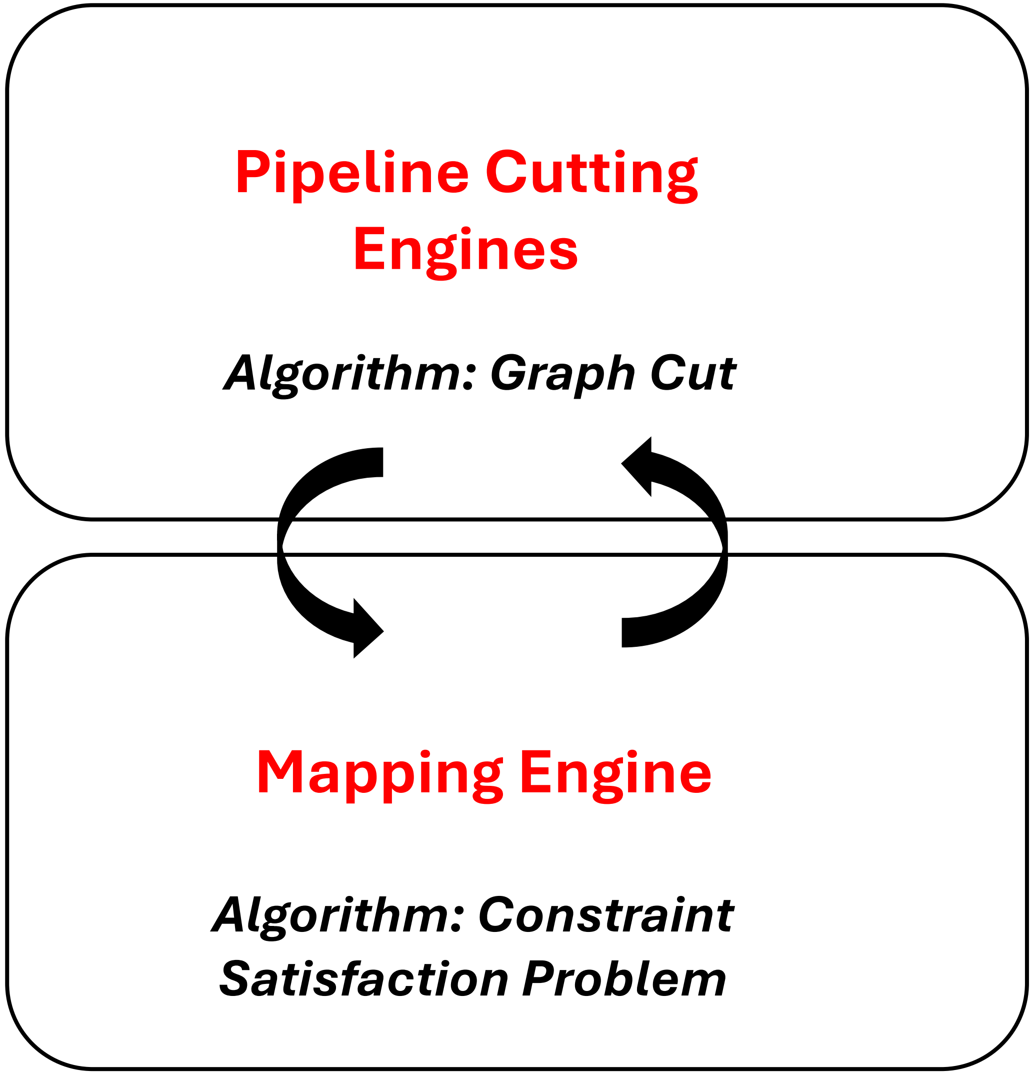
Algorithm: Graph Cut

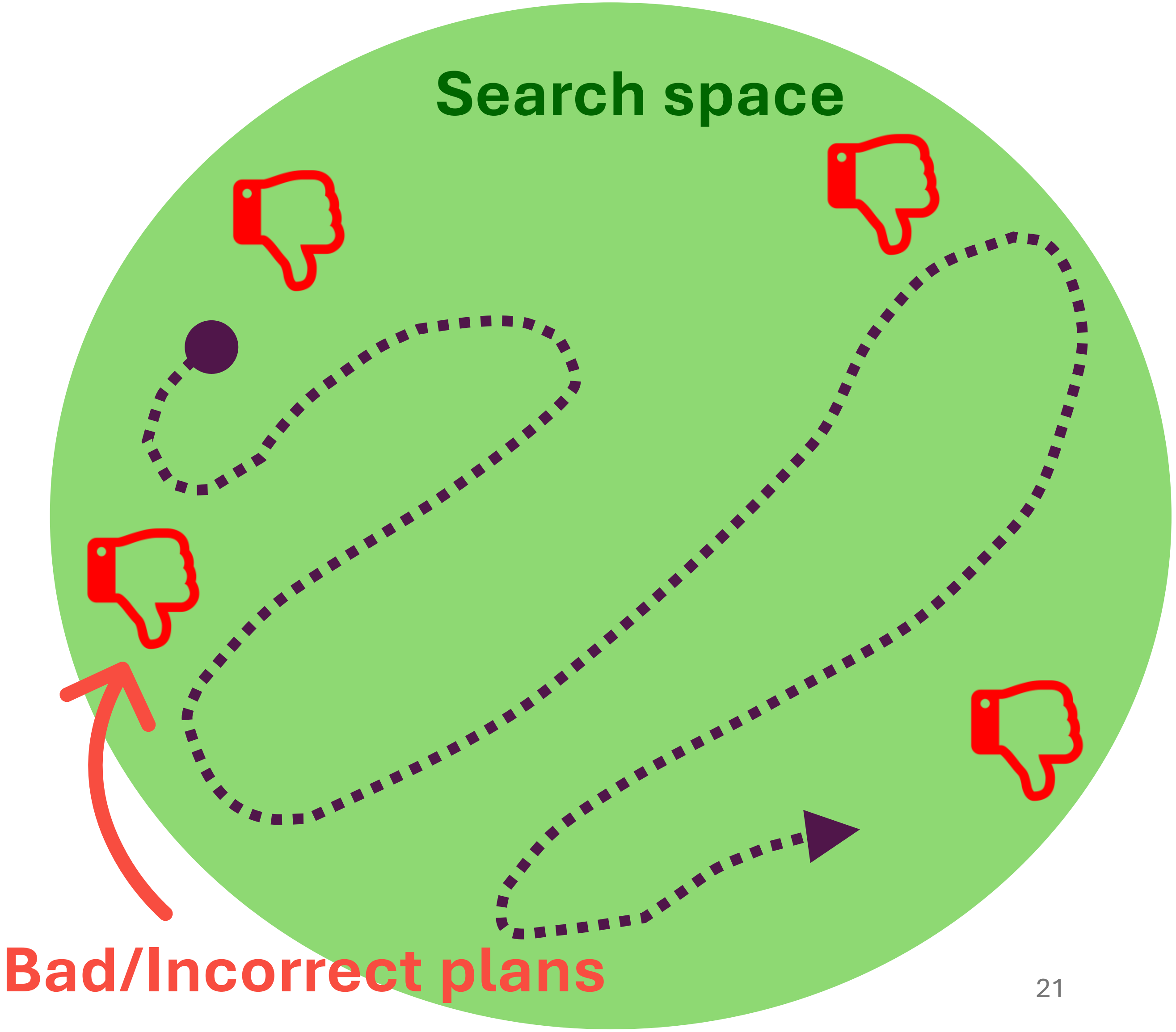
Mapping Engine

Algorithm: Constraint Satisfaction Problem

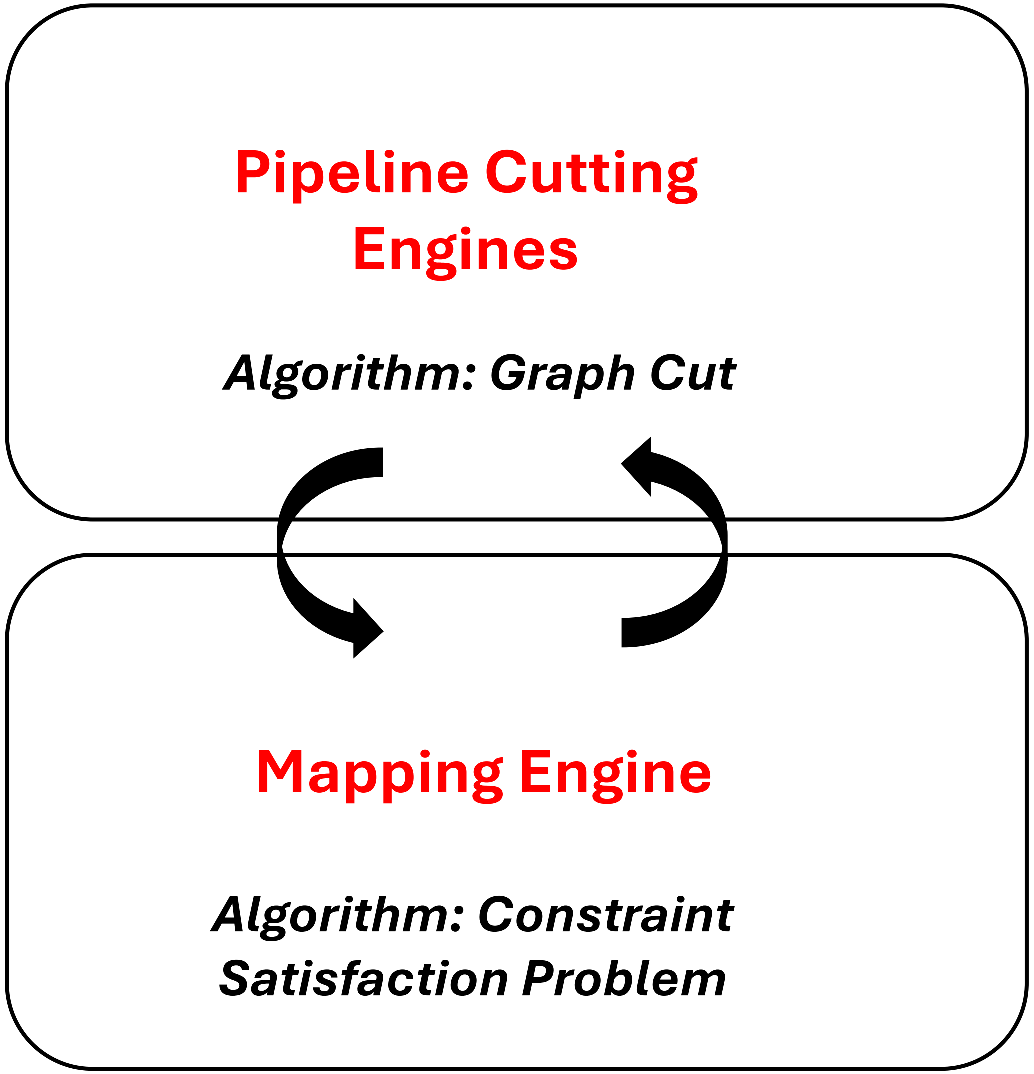


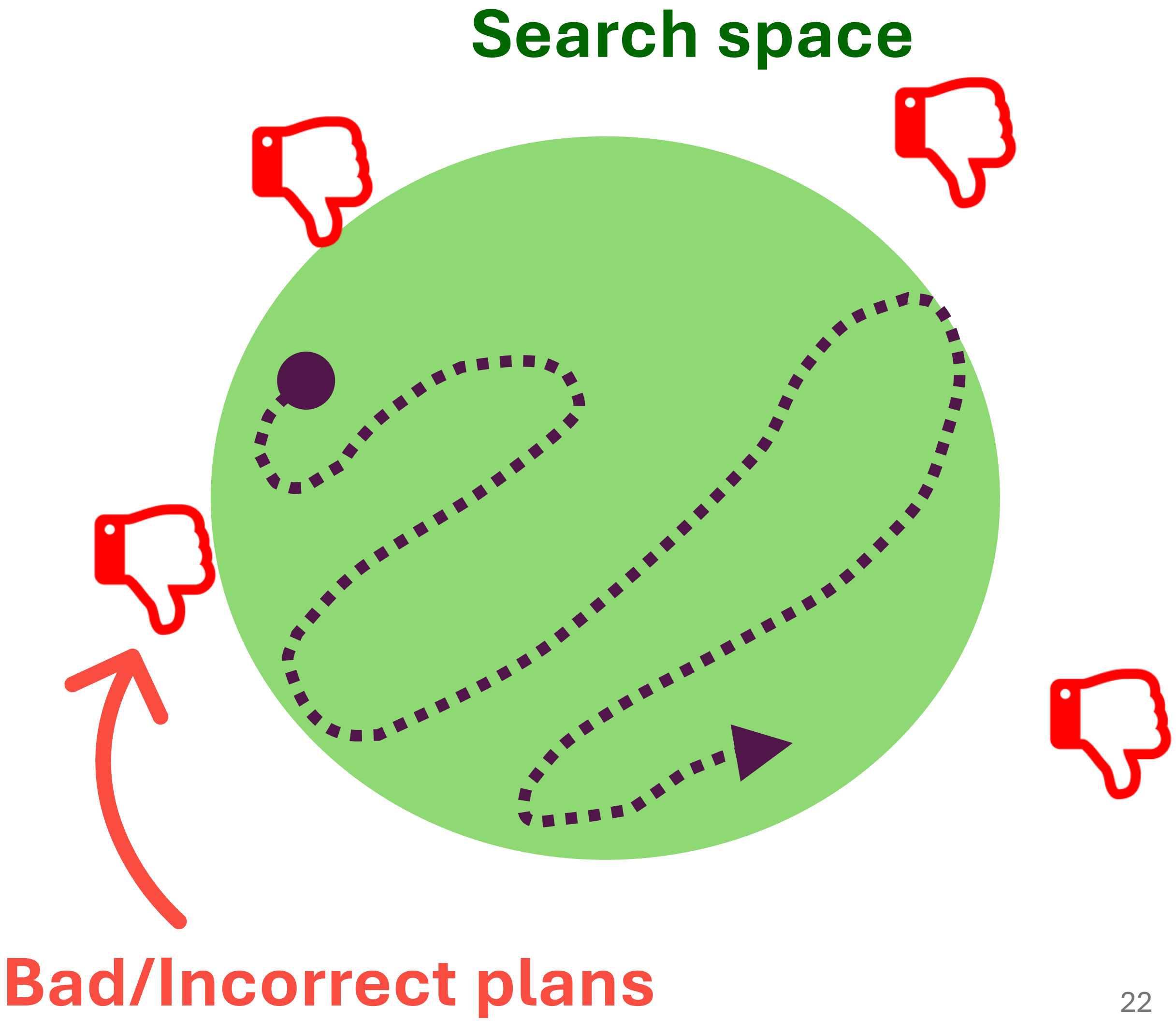
Adding Constraints to Improve Efficiency



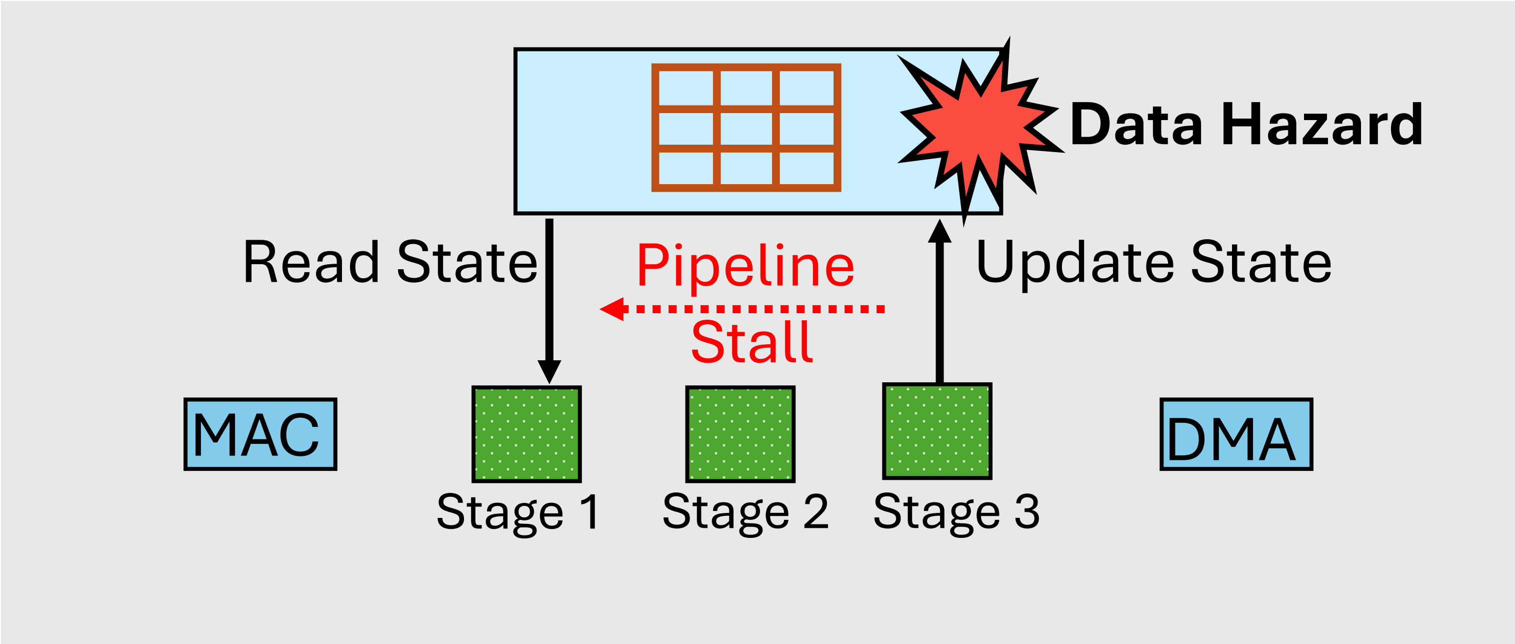


Adding Constraints to Improve Efficiency





Example of a Bad Pipeline Plan



Prune this bad plan: pipeline cut should avoid splitting state

Alkali Framework

- **C frontend, compiler**
  - 20K lines C++ using MLIR
  - Compiler opts: peephole, CSE, DCE, copy to zero-copy..
- **Four NIC backends**
  - Agilio (on-path SoC): MicroC
  - BlueField-2 (off-path SoC): LLVM ARM binary
  - Alveo (FPGA) : Verilog RTL
  - PANIC (ASIC NIC): LLVM RISCV binary
- **Runtime libraries for each NIC**
  - Event controller
  - Inter-compute unit communication queues

https://github.com/utnslab/Alkali

Alkali Roadmap

Alkali Compiler and IR (May 2025)

Alkali Runtime and Libraries (Aug/Sep 2025)

Enable End-to-End flow on CPU/FPGA targets

dev guides for IR, optimization and backends

Feature Development (Late 2025)

P4 Front/Backend

BF3 DPA Backend

Functional Simulation

…

Alkali Roadmap – P4 Front/Backend

P4HIR: Towards Bridging P4C with MLIR

P4 Frontend: Integrate with P4 within MLIR ecosystem

- **Leverage the P4HIR Project**
- **Translation as Dialect Conversion**

P4 Backend: Transformations for semantically compatible with P4

[P4HIR] https://github.com/p4lang/p4mlir-incubator

Alkali Roadmap

Alkali Compiler and IR
(May 2025)

Alkali Runtime and Libraries
(Aug/Sep 2025)
Enable End-to-End flow on CPU/FPGA targets
dev guides for IR, optimization and backends

Compiler Infrastructure Cleanup:
Compossibility and Extensibility
(Current)

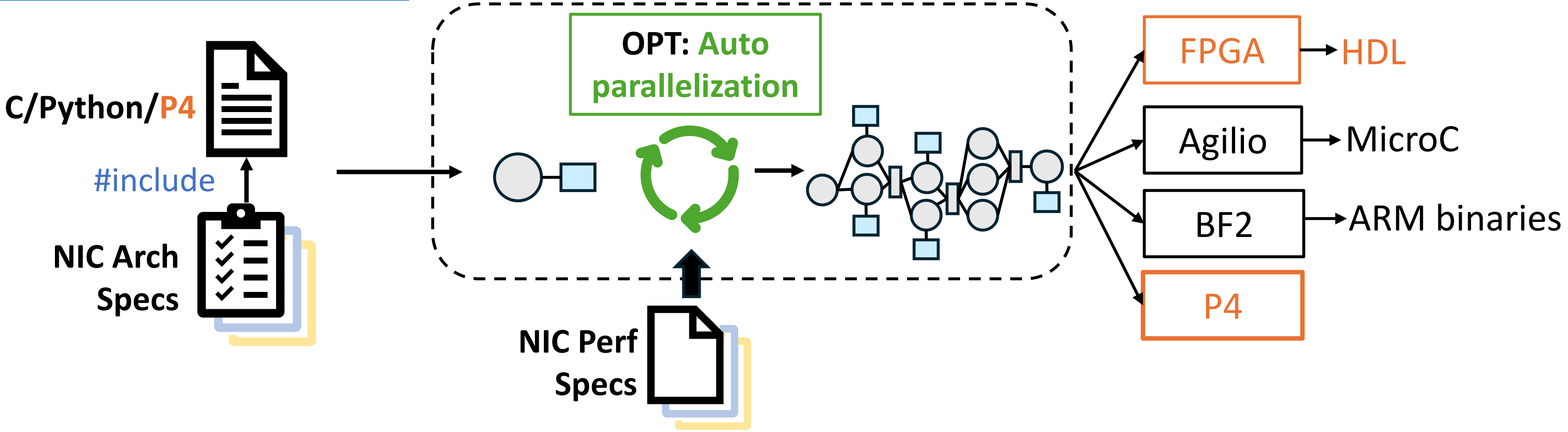
Feature Development
(Late 2025)
P4 Front/Backend
BF3 DPA Backend
Functional Simulation
…

Using Alkali – Compositable Infrastructure

Programming: Single threaded event processing

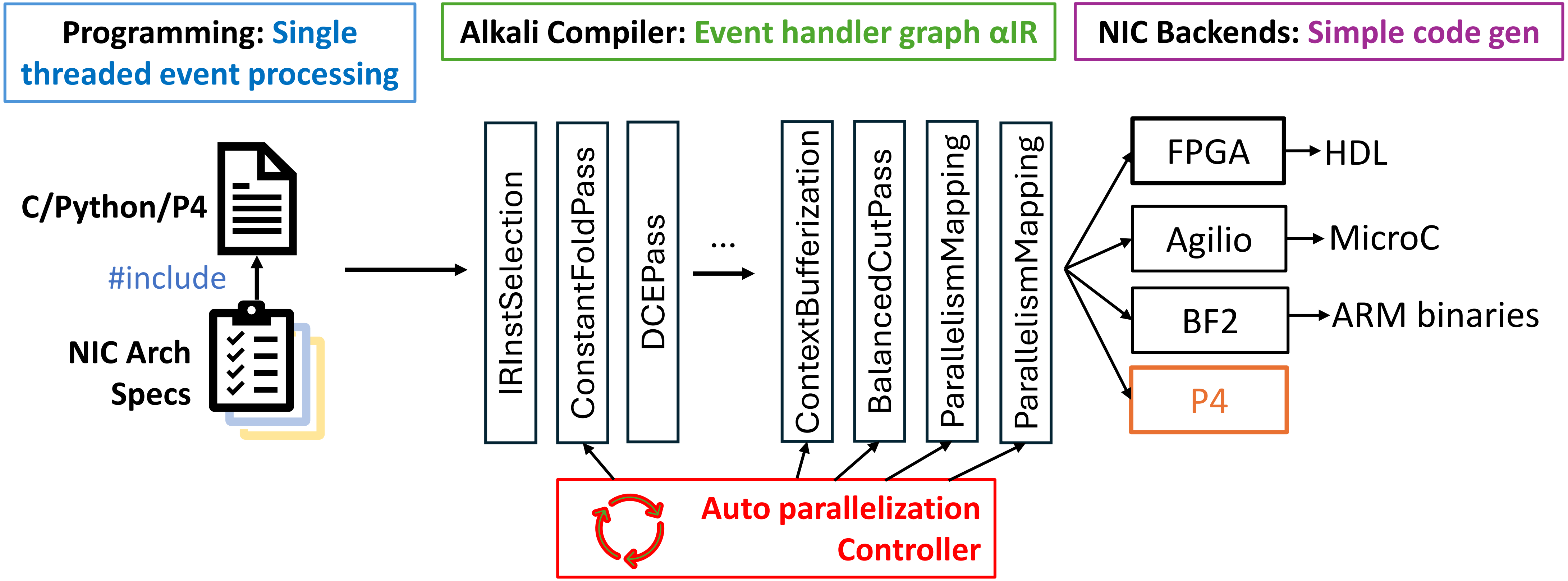
Alkali Compiler: Event handler graph αIR

NIC Backends: Simple code gen



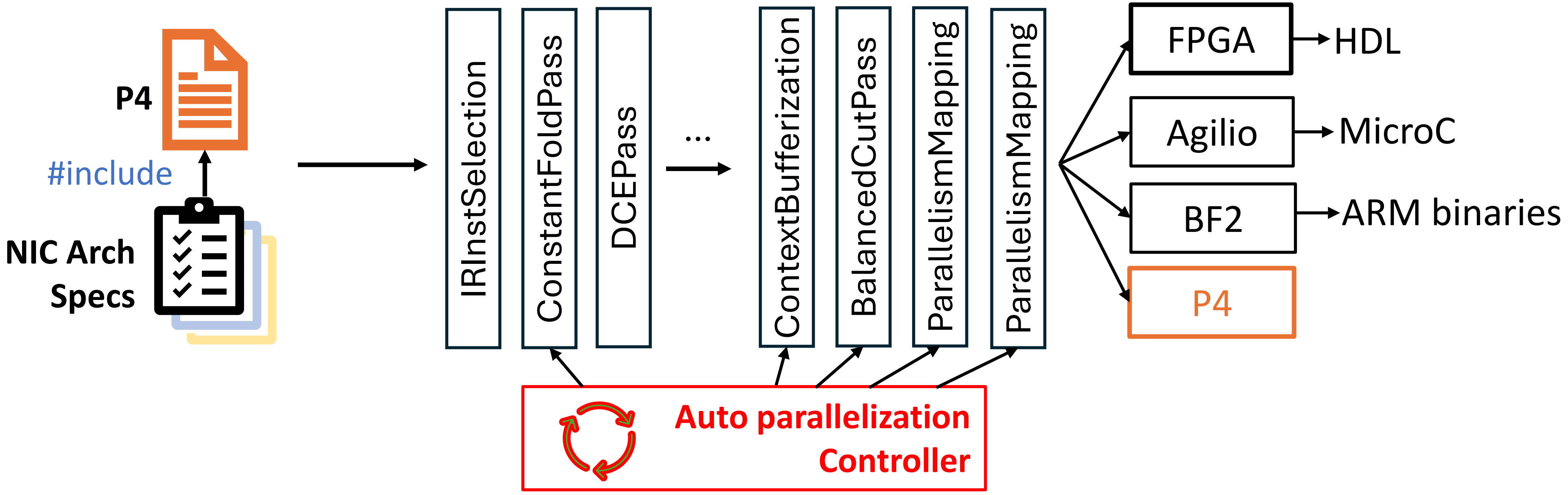
Using Alkali – Compositable Infrastructure

- **Alkali components** could be composed for customized flow



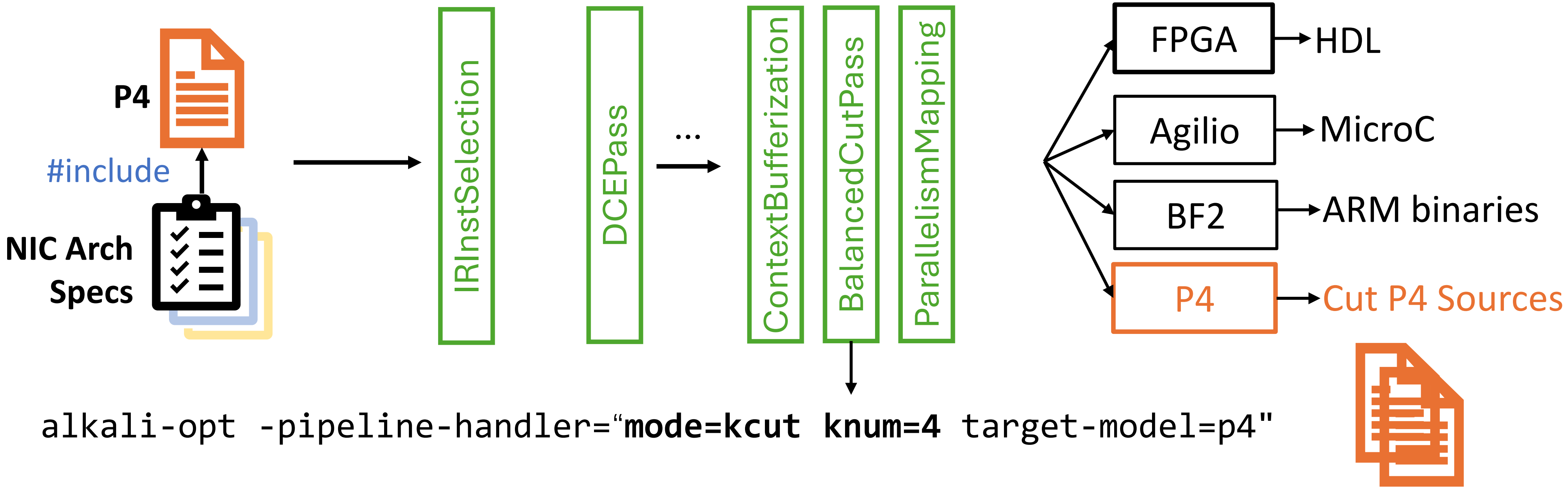
Using Alkali – Compositable Infrastructure

- **Alkali components** could be composed for customized flow
- **Example: Alkali as P4-to-P4 transpiler (pipeline cut)**



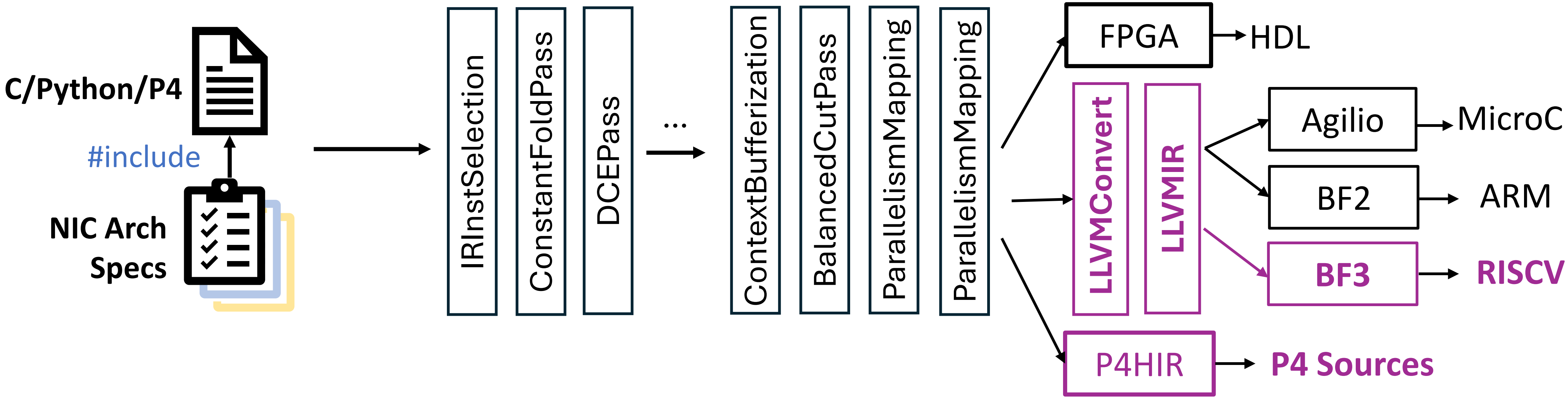
Using Alkali – Compositable Infrastructure

- **Alkali components** could be composed for customized flow
- **Example: Alkali as P4-to-P4 transpiler (pipeline cut)**



Using Alkali – Extensible Infrastructure

- **Alkali IR** as interface for optimizations
- **Plug 'n Play** for Frontend, Compiler and New Backends extension



Conclusion

- **Key Idea:** Use an intermediate representation (IR) to abstract the compute parallelism and state access patterns of NIC programs.

- Leverage this IR to build a reusable compiler framework with optimizations that enable automated parallelization.

https://github.com/utnslab/Alkali