Core C++ 2021

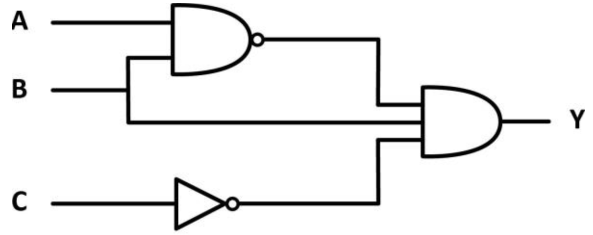
Design Patterns for Hardware Packet Procesing on FPGAs

Haggai Eran

THE ACADEMIC of TEL AVIV-YAFFO

TL;DR

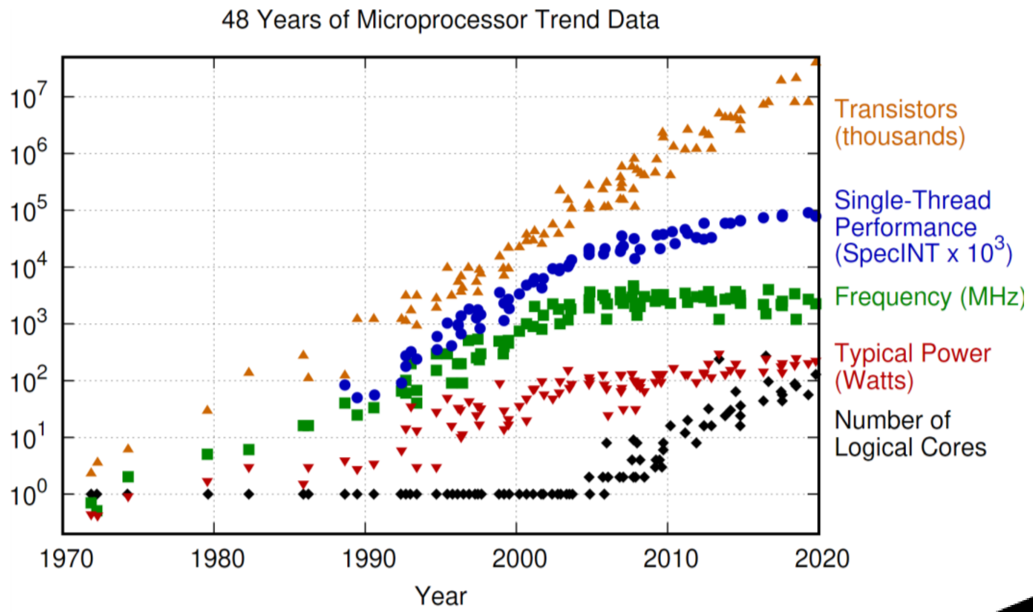
- C++ for FPGA hardware design
- Packet processing example
- Design patterns and building blocks library



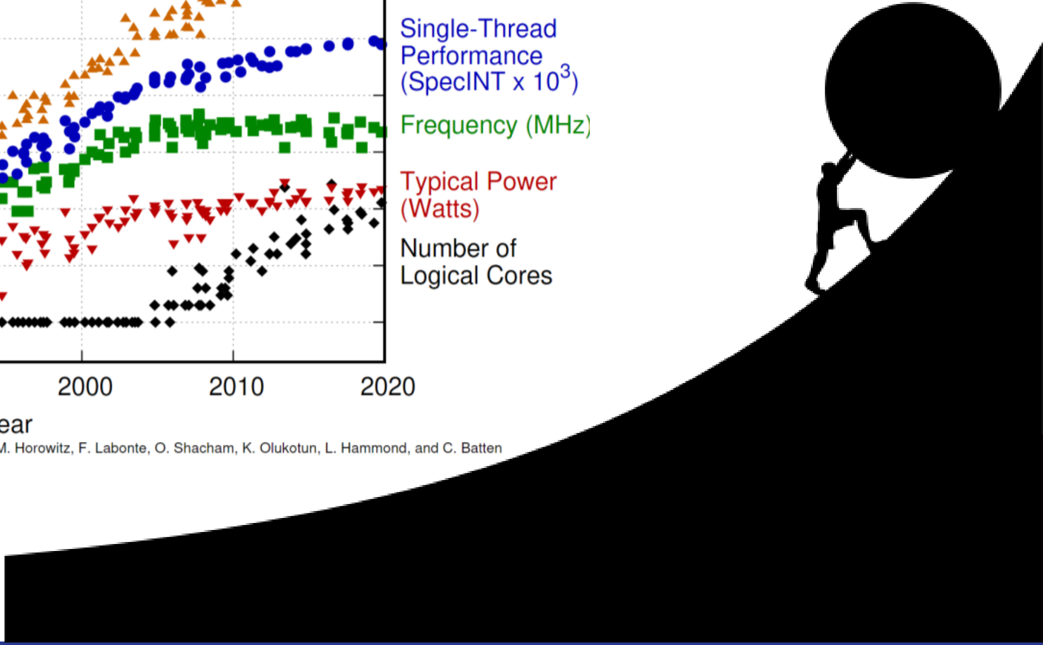
H. Eran, L. Zeno, Z. István and M. Silberstein,
"Design Patterns for Code Reuse in HLS Packet Processing Pipelines,"
2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019

NVIDIA NETWORKING TECHNION Israel Institute of Technology institute imdea software

Motivation: end of Dennard Scaling

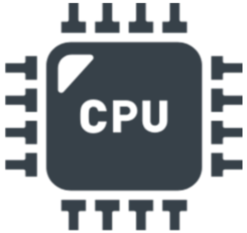


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

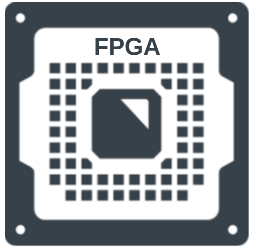


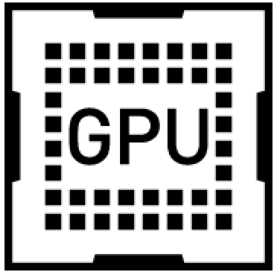
https://zenodo.org/record/3947824#.YSHXQI5vYuU

Accelerators & heterogenous computing

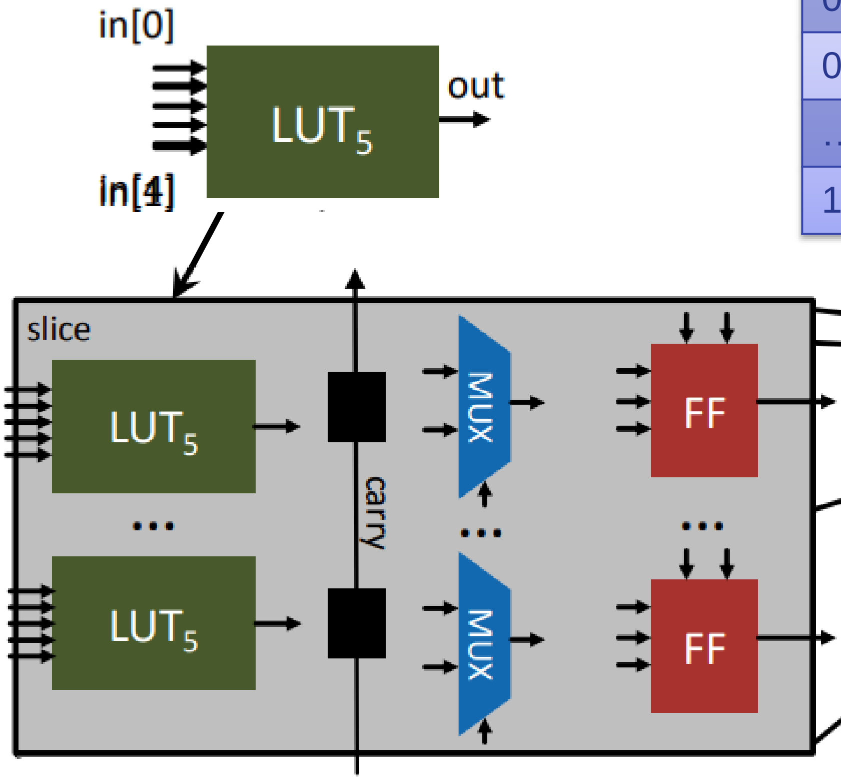


Cloud TPU v3

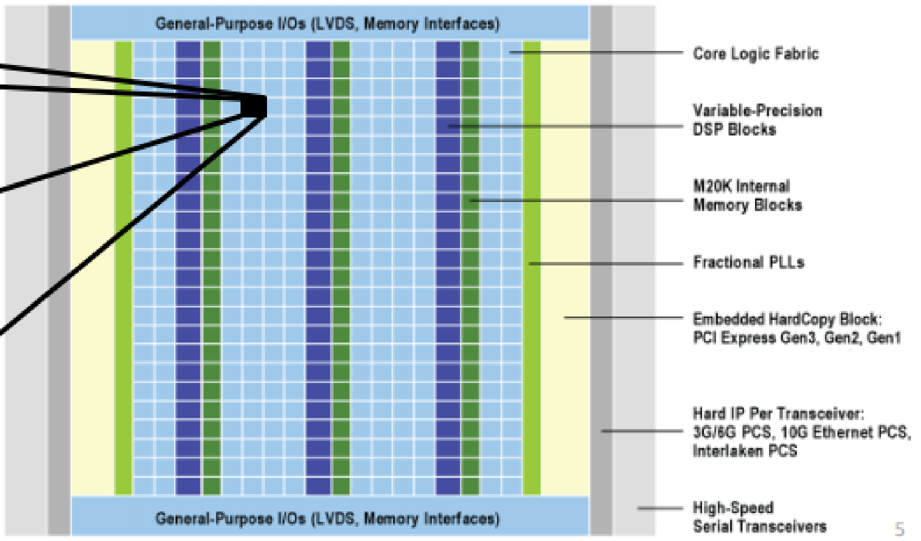




What are FPGAs?



Input	Value
00000	0
00001	0
…	…
11111	1



General-Purpose I/Os (LVDS, Memory Interfaces)
Core Logic Fabric
Variable-Precision DSP Blocks
M20K Internal Memory Blocks
Fractional PLLs
Embedded HardCopy Block: PCI Express Gen3, Gen2, Gen1
Hard IP Per Transceiver: 3G/6G PCS, 10G Ethernet PCS, Interlaken PCS
High-Speed Serial Transceivers
General-Purpose I/Os (LVDS, Memory Interfaces)

Network packet processing & FPGAs

- High-throughput
- Low latency
- Predictability
- Flexibility

E.g.

- AccelNet on Microsoft Azure [Firestone et al. NSDI'18]

Network packet processing on FPGAs programming alternatives

Efficiency/
expressiveness

Simplicity

Register Transfer
Language (RTL)

High-level synthesis

Domain specific
languages

Verilog HDL

VHDL

P4

High-level synthesis (HLS)

$$\begin{tikzcd}\text{High-level code (C/C++/OpenCL)} \arrow[r] & \text{RTL (Verilog)} \arrow[r] & \text{FPGA bitstream}\end{tikzcd}$$

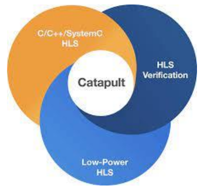
- 👍 Abstract underlying hardware details
  - Reuse a design on different hardware
- 👍 Rapid simulations
- 👍 Design space exploration

intel HLS COMPILER

Vivado™ HLS

SYNOPSYS®

Symphony C



Mentor® A Siemens Business

cadence

Stratus

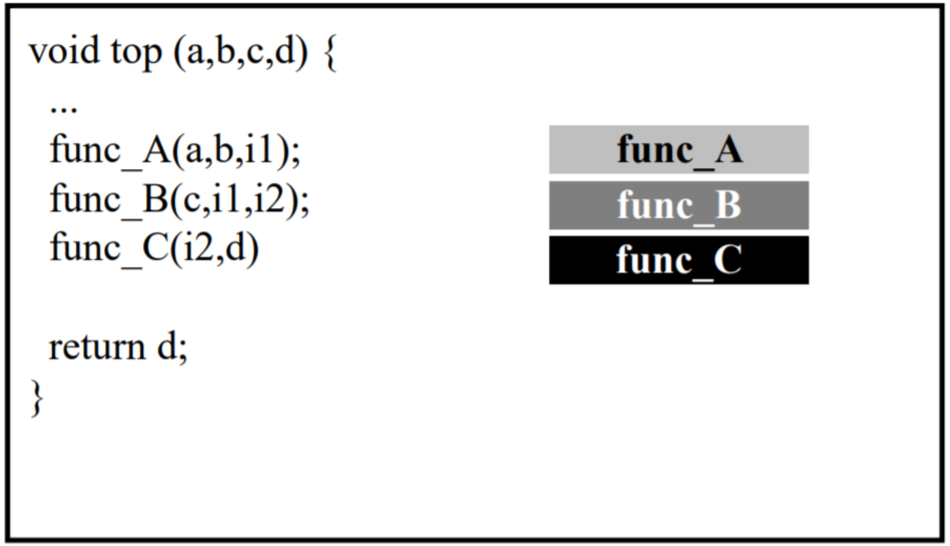
Why is it hard to build an HLS networking lib?

- 👎 Only a subset of C++ is synthesizable.
  - No virtual functions
  - No dynamic memory allocation
  - No casting pointers
- 👎 Vivado HLS 2018.2 supports C++11
- 👎 Strict interfaces and patterns for performance.

In this talk

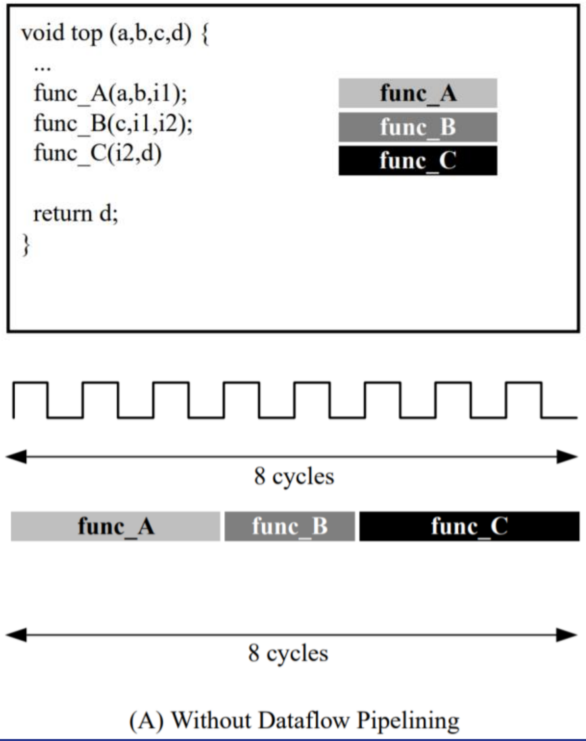
- Vivado HLS background
- Legacy HLS – how others have used HLS for high-performance networking, limiting reuse
- ntl – template library for HLS packet processing and methodology used to develop it
- Quantitative comparison

Vivado HLS's dataflow optimization



```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

Vivado HLS's dataflow optimization



```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

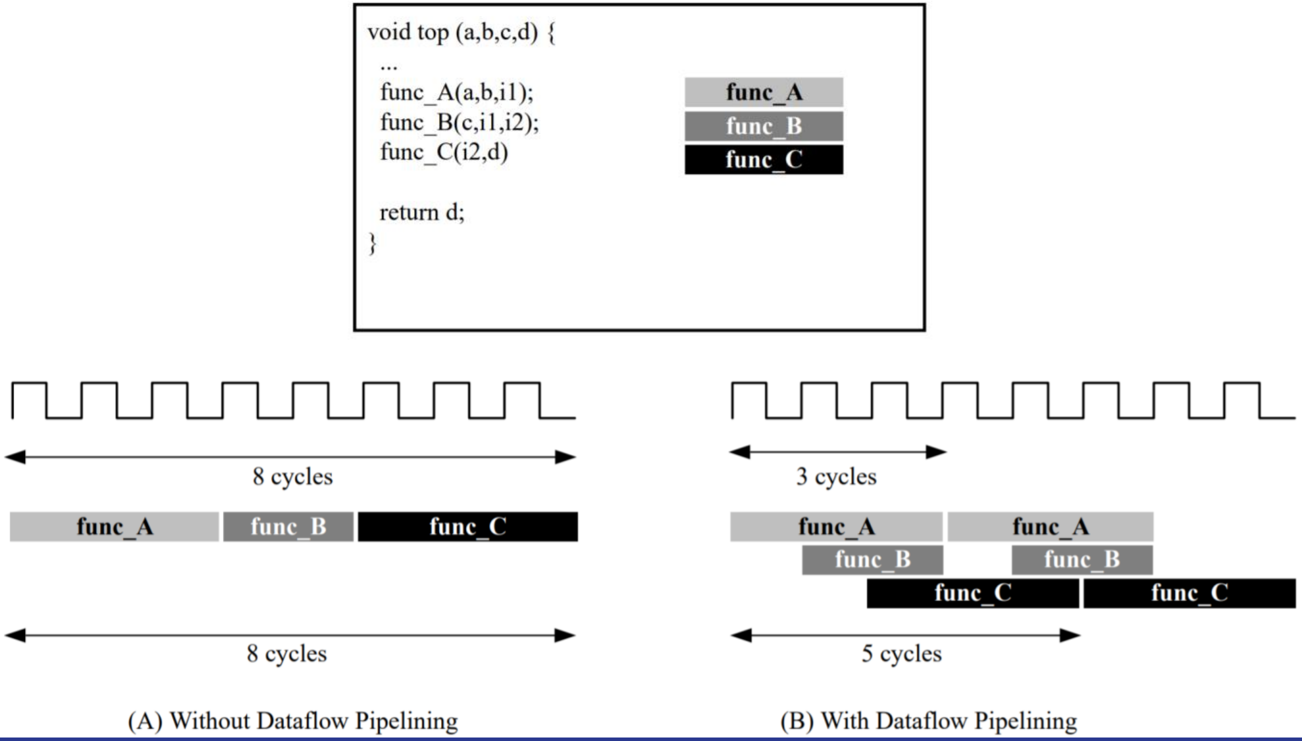
(A) Without Dataflow Pipelining

[Vivado Design Suite User Guide – High-Level Synthesis]

Vivado HLS's dataflow optimization

Figure 64: **Dataflow Optimization**

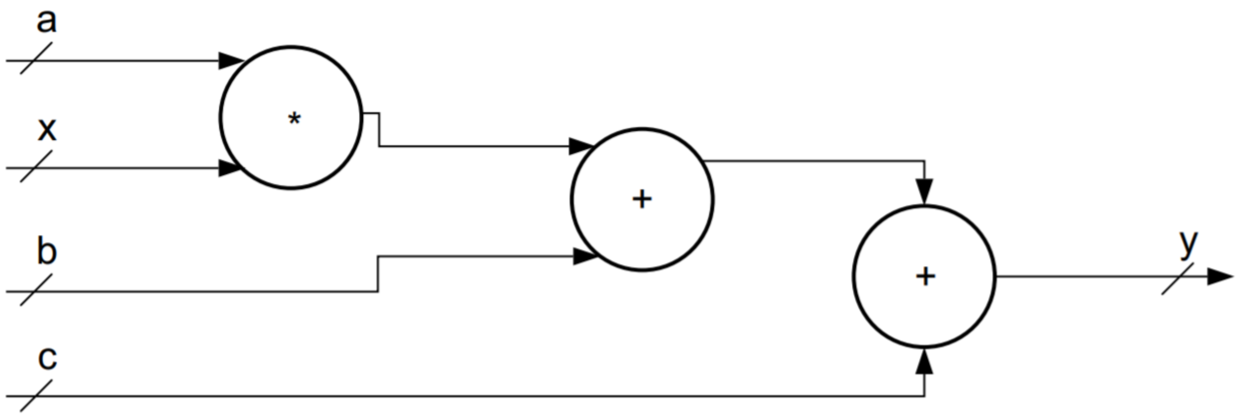
```
#pragma HLS dataflow
```



Vivado Design Suite User Guide – High-Level Synthesis

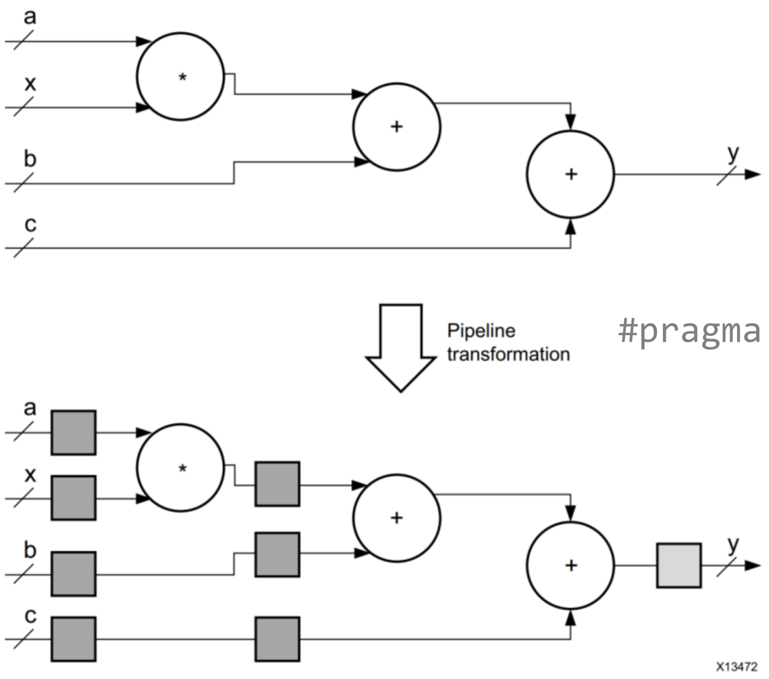
Vivado HLS’s pipeline optimization

$$y = (a \times x) + b + c$$



Vivado HLS's pipeline optimization

$$y = (a \times x) + b + c$$



```
#pragma HLS pipeline
```

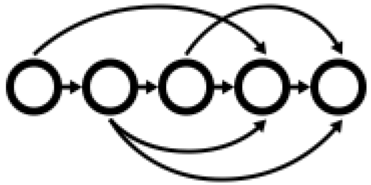
Figure 2-9: **FPGA Implementation of a Compute Function**

Legacy HLS methodology

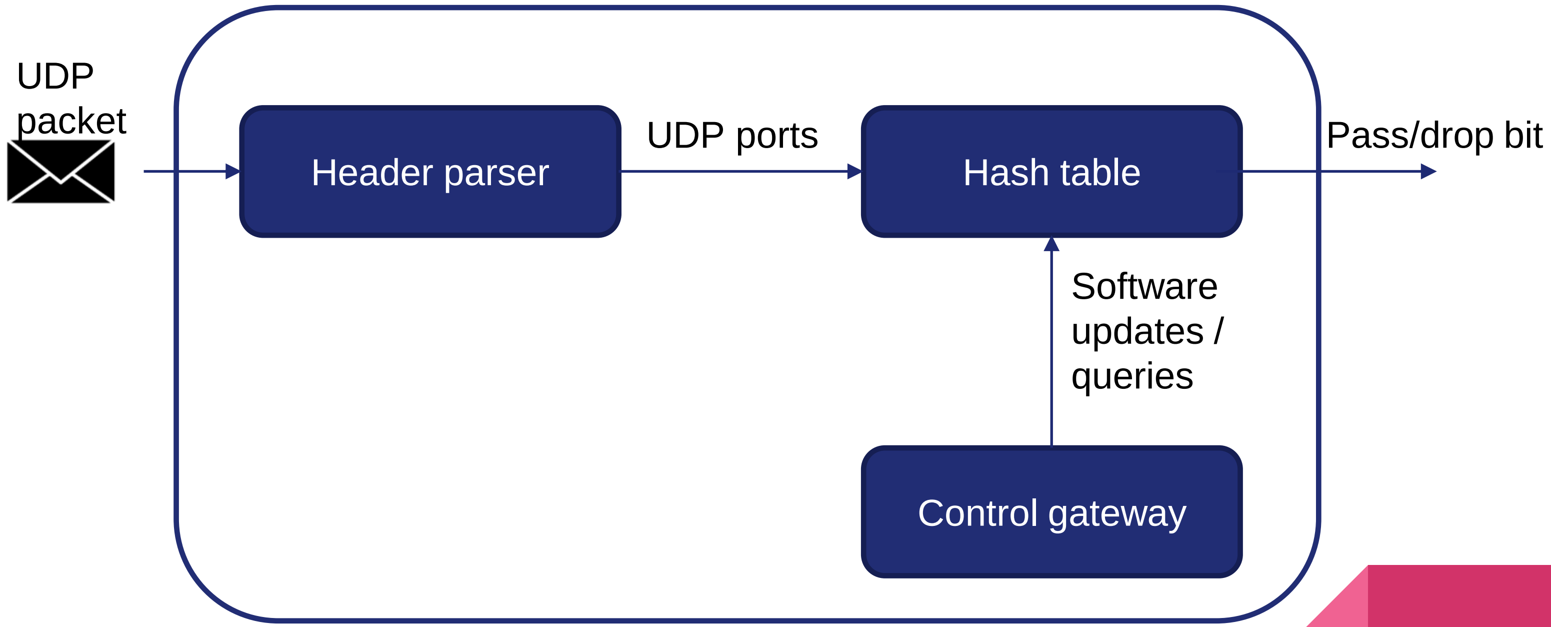
Legacy HLS - how is HLS used for packet processing?

Data-flow design

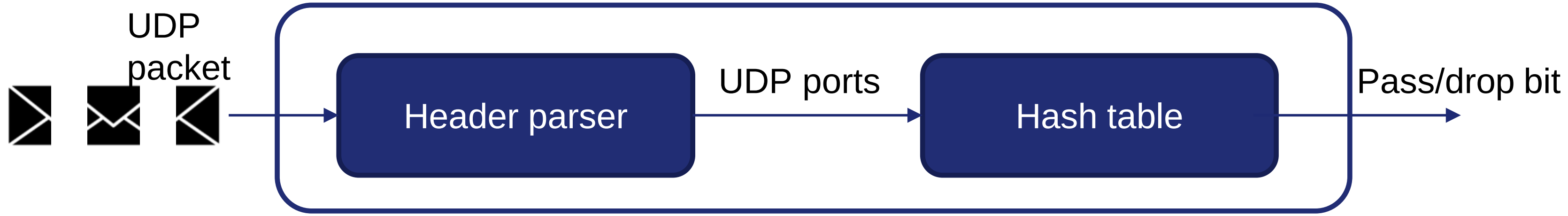
- A fixed graph of independent elements
- Operate on data when inputs are ready
- Examples: [Blott ’13], [XAPP1209 ‘14], [Sidler ’15], ClickNP [Li ’16].



Running example: UDP stateless firewall



Packet interface: flits



Legacy HLS: data-flow in Vivado HLS

```
struct axi_data {
        ap_uint<256> data;
        ap_uint<32> keep;
        ap_uint<1> last;
};

typedef hls::stream<axi_data> axi_data_stream;
typedef hls::stream<ap_uint<1>> bool_stream;
```

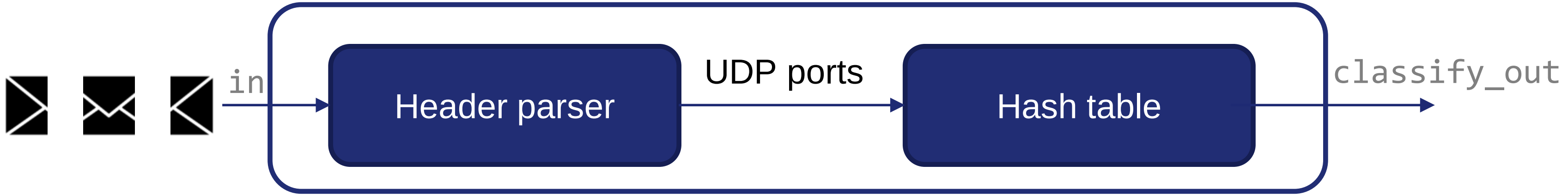
Describe arbitrary precision integers.

Synthesized as FIFO.

Legacy HLS: data-flow in Vivado HLS

```
void firewall_top(axi_data_stream& in, bool_stream& classify_out);
```

Passing by reference synthesizes a bus interface; direction is inferred



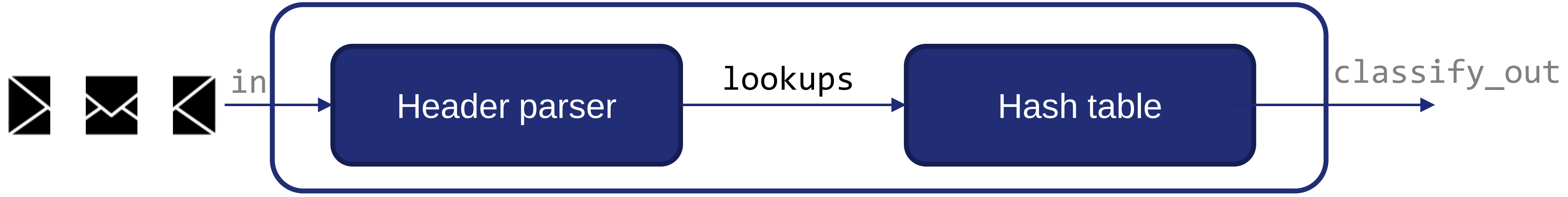
Legacy HLS: data-flow in Vivado HLS

```
void firewall_top(axi_data_stream& in, bool_stream& classify_out)
{
#pragma HLS dataflow
    static hls::stream<hash_tag> lookups;

    parser(in, lookups);
    hash_table(lookups, classify_out /* ... */);
}
```

Compiler directive for the data-flow optimization

static used to describe the module's internal state

Function invocation = hardware module instantiation



Legacy HLS: simple parser state-machine

```
void parser(axi_data_stream& in, hls::stream<hash_tag>& out)
{
#pragma HLS pipeline
    static enum { IDLE, FIRST, REST } state = IDLE;
    static hash_tag ret;

    axi_data flit;

…
```

Accept the next flit while processing previous ones

Where we are within the packet

Output may be built over multiple invocations

Legacy HLS: simple parser state-machine

```
void parser(axi_data_stream& in, hls::stream<hash_tag>& out) {
…
    switch (state) {
    case IDLE:
        if (in.empty() || out.full())
            return;

        in.read_nb(flit);
        ret = extract_headers_flit1(flit);
        state = flit.last ? IDLE : FIRST;
        if (flit.last)
            out.write_nb(ret);
        break;
…
```

First flit of the packet

Check for available inputs, output space

Update internal state from input

Output

Legacy HLS: issues

- Static variables make it difficult to reuse code
- Limited use of classes
- Repetitive code handling streams

*There are three great virtues of a programmer: **Laziness**, Impatience and Hubris.*

Larry Wall

Creator of the Perl programming language

https://www.flickr.com/photos/vadim_trochinsky/9573703858/

ntl HLS methodology

How to build reusable data-flow element pattern?

- Basic elements
  - C++ objects for each data-flow element
  - State kept as member variables
  - `step()` method implements functionality
  - Inline methods embedded in the caller
  - All interfaces are `hls::stream` (members/parameters)
- Reuse with customization via function objects / lambdas
- Composed through aggregation: reusable sub-graph.

Networking Template Library (`ntl`)

Class library of packet processing building blocks.

Category	Classes
Header processing elements	`pop/push_header, push_suffix`
Data-structures	`array, hash_table`
Scheduler	`scheduler`
Basic elements	`map, scan, fold, dup, zip, link`
Specialized stream wrappers	`pack_stream, pfifo, stream<Tag>`
Control-plane	`gateway`

Networking Template Library (`ntl`)

Class library of packet processing building blocks.

Category	Classes
Header processing elements	`pop/push_header`, `push_suffix`
Data-structures	`array`, `hash_table`
Scheduler	`scheduler`
Basic elements	`map`, **`scan`**, **`fold`**, `dup`, `zip`, `link`
Specialized stream wrappers	`pack_stream`, **`pfifo`**, `stream<Tag>`
Control-plane	`gateway`

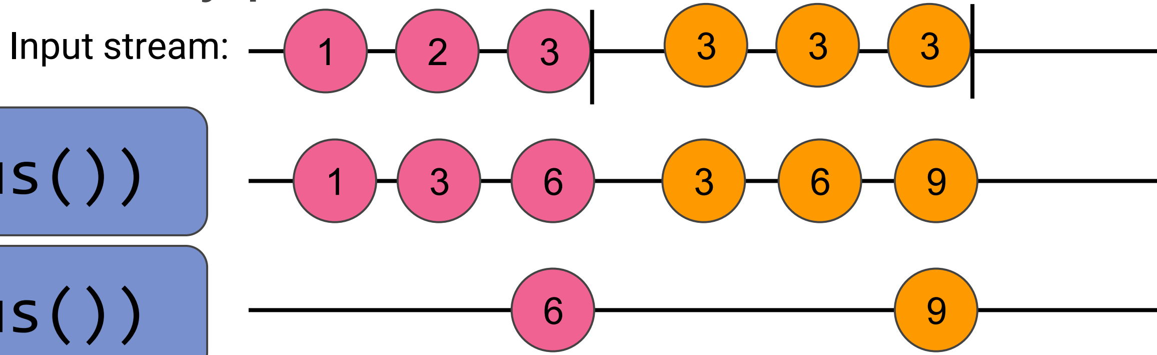
Example: scan and fold

Common operators in functional and reactive programming.

Modified to reset state for every packet.

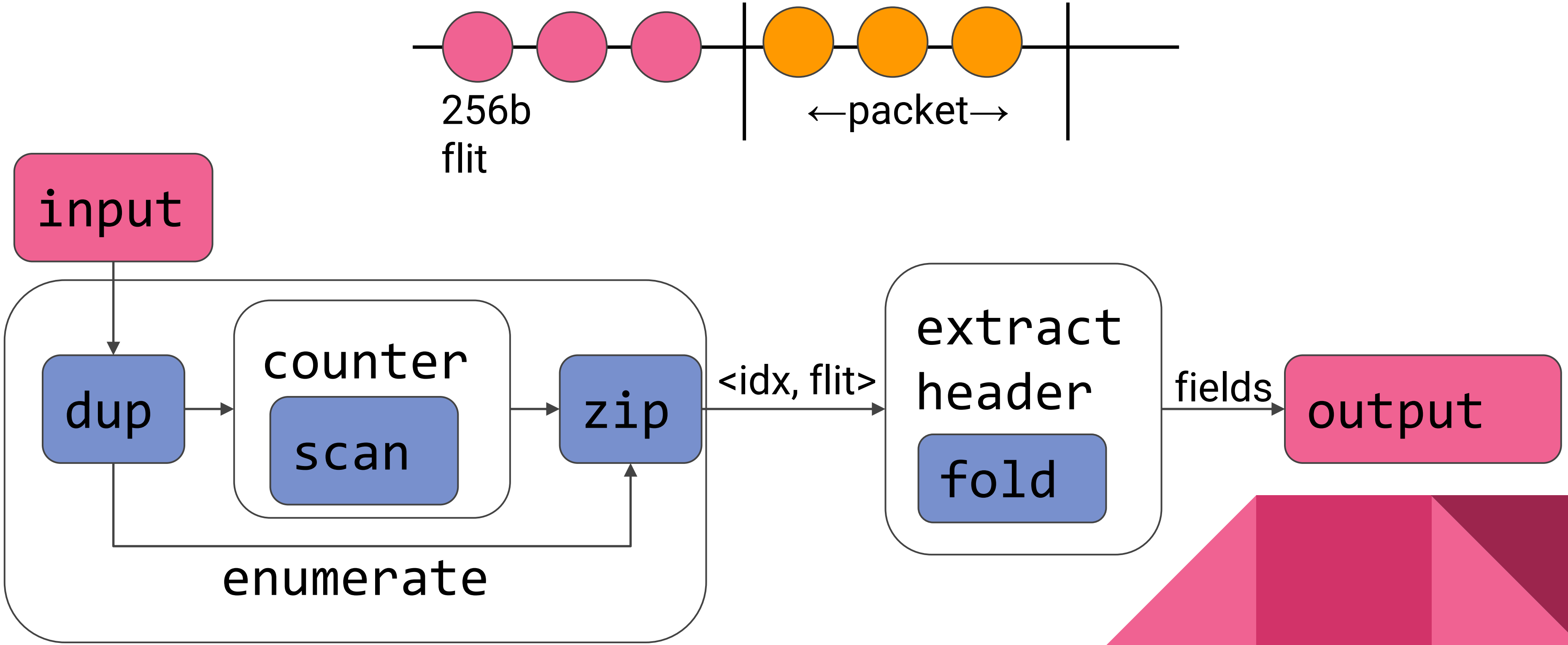
```
scan.step(input, plus())
```

```
fold.step(input, plus())
```



Reduce boilerplate hls::stream handling

Fold & scan usage: parser example



Parser class with ntl

```
class parser {
public:
    void step(axi_data_stream& in);
    hls::stream<hash_tag> out;

private:
    ntl::enumerate<ntl::axi_data> _enum;
    extract_metadata _extract;
};
```

Compiler synthesizes the module from the step function

We can instantiate output FIFO within the class to simplify its use

Instantiate sub-modules

Parser step function

```
void parser::step(axi_data_stream& in)
{
#pragma HLS dataflow
    _enum.step(in);
    _extract.step(_enum.out);
    ntl::link(_extract.out, out);
}
```

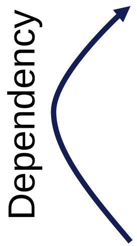
Top function wrapper

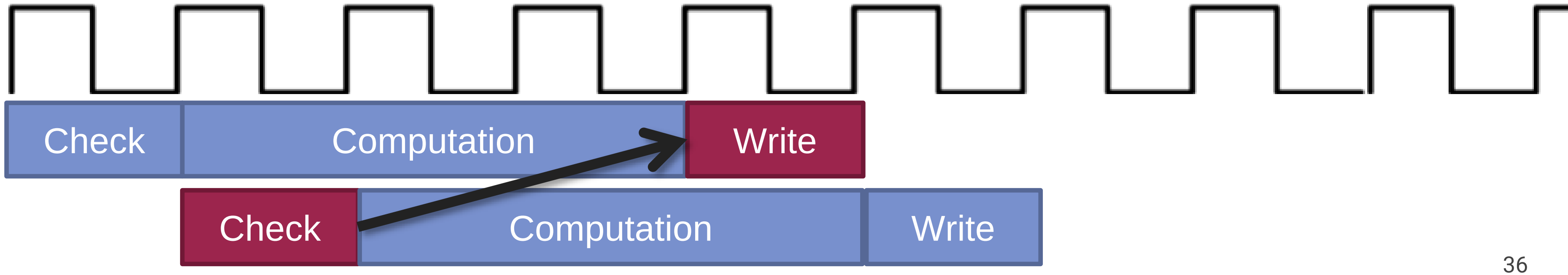
```
void firewall_top(axi_data_stream& in, bool_stream& classify_out)
{
#pragma HLS dataflow
    static firewall f;

    f.step(in, classify_out);
}
```

static only used once in the top function wrapper

Pipeline dependencies

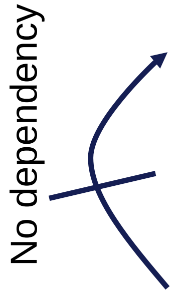
```
#pragma HLS pipeline

if (out.full())
    return;

auto ret = N_cycle_computation(…)
out.write_nb(ret);
```

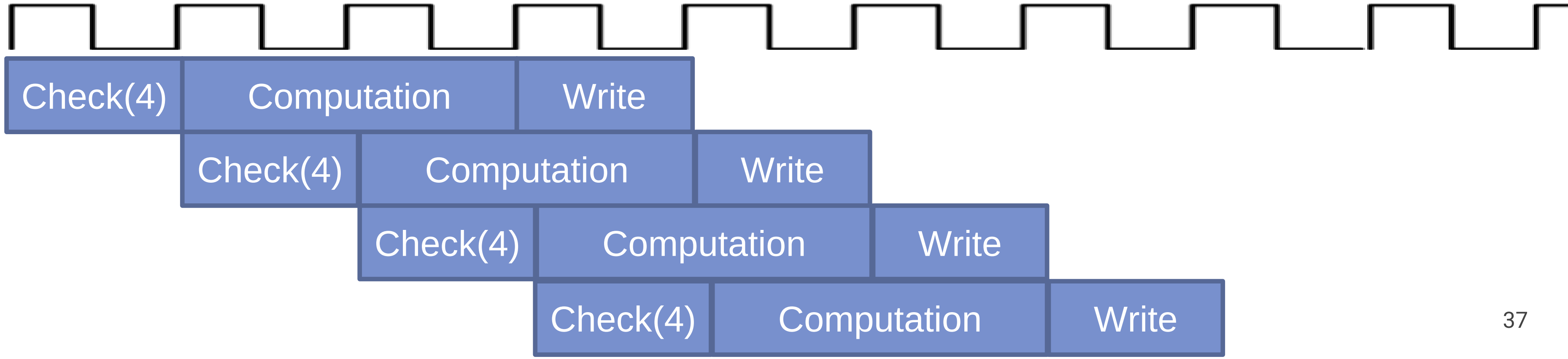




Pipeline dependencies

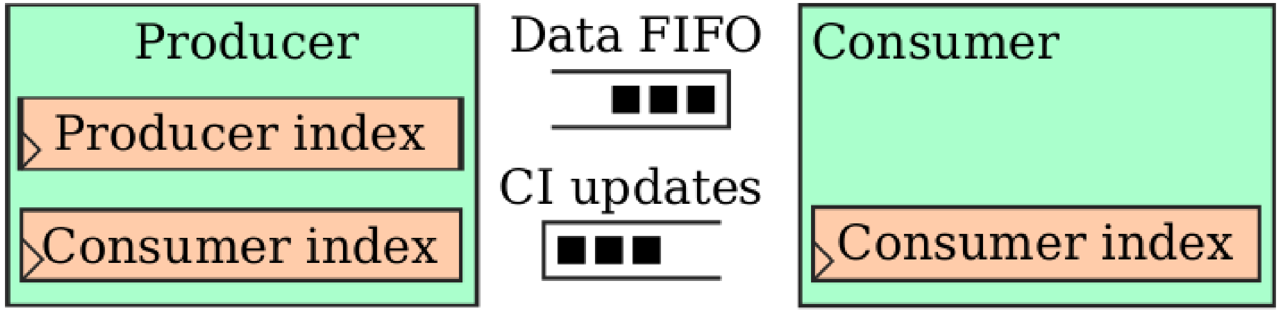
```
#pragma HLS pipeline

if (out.has_room_for(N+1))
    return;

auto ret = N_cycle_computation(…)
out.write_nb(ret);
```





Programmable-threshold FIFO

`hls::stream` replacement



Programmable-threshold FIFO

```
template <typename T, size_t stream_depth>
class programmable_fifo {
public:
    bool write_nb(const T& t);
    bool read_nb(T& t);
    bool full();
    bool empty();
private:
    /* producer and consumer state */
    …
    hls::stream<T> _stream;
}
```

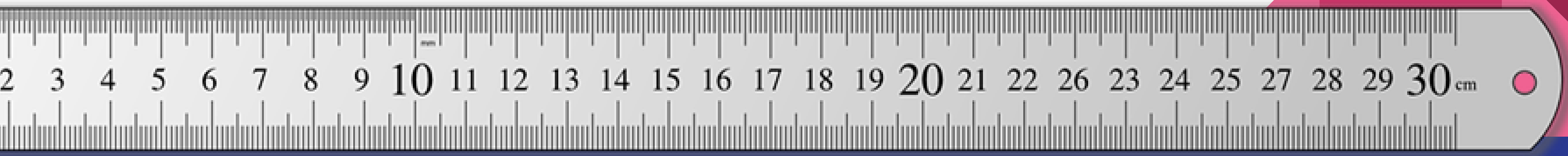
Use compiler directive to inline into callers

Evaluation

- How does `ntl` compare against legacy HLS, P4?
- Can we build a relatively complex application with `ntl`?

Targeting Mellanox Innova Flex SmartNIC

- Xilinx Kintex UltraScale XCKU060 FPGA
- 216.25 MHz clock rate



Stateless UDP firewall example

Use on-chip hash-table to classify packets.

	Thpt.	Latency	LUTs	FFs	BRAM	LoC
HLS/`ntl`	72 Mpps	25 cycles	5296	7179	12	218
HLS legacy	72 Mpps	16 cycles	4087	4287	12	593
P4 (SDNet 2018.2)	108 Mpps	211 cycles	34531	49042	193	92

Stateless UDP firewall example

Use hash-table to classify packets.

All exceed line rate (59.5 Mpps)

	Thpt.	Latency	LUTs	FFs	BRAM	LoC
HLS/`ntl`	72 Mpps	25 cycles	5296	7179	12	218
HLS legacy	72 Mpps	16 cycles	4087	4287	12	593
P4 (SDNet 2018.2)	108 Mpps	211 cycles	34531	49042	193	92

Stateless UDP firewall example

Use hash-table to classify packets.

x2.7 less lines of code compared to legacy

	Thpt.	Latency	LUTs	FFs	BRAM	LoC
HLS/`ntl`	72 Mpps	25 cycles	5296	7179	12	218
HLS legacy	72 Mpps	16 cycles	4087	4287	12	593
P4 (SDNet 2018.2)	108 Mpps	211 cycles	34531	49042	193	92

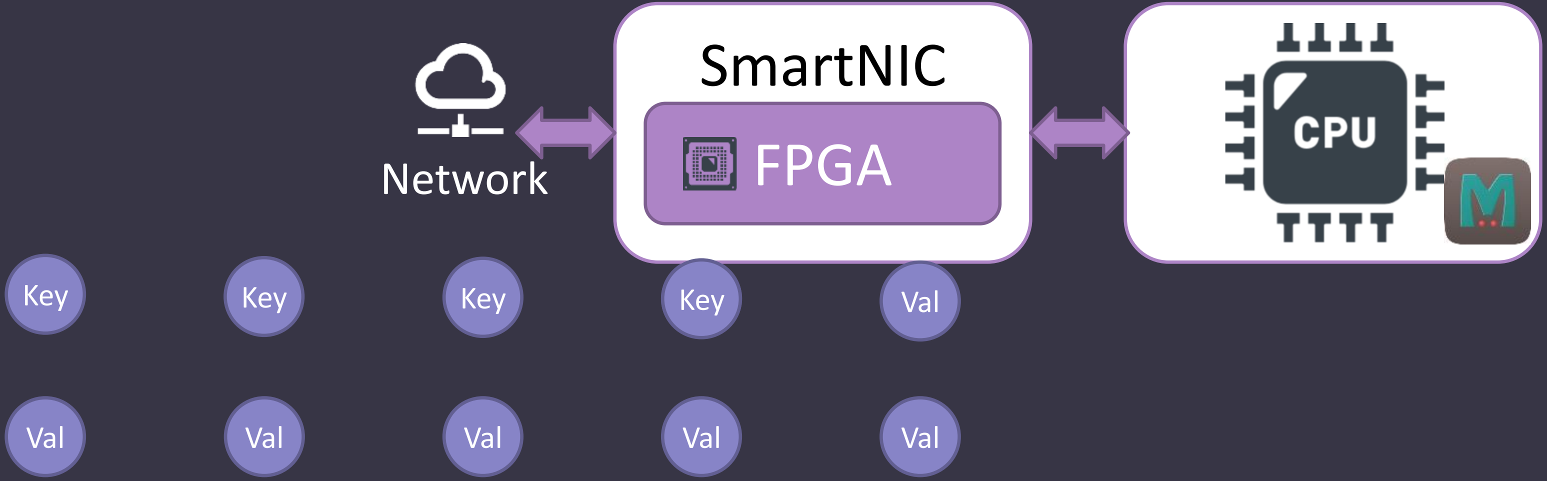
Stateless UDP firewall example

Use hash-table to classify packets.

	Thpt.	Latency	LUTs	FFs	BRAM	LoC
HLS/`ntl`	72 Mpps	25 cycles	5296	7179	12	218
HLS legacy	72 Mpps	16 cycles	4087	4287	12	593
P4 (SDNet 2018.2)	108 Mpps	211 cycles	34531	49042	193	92

`ntl` requires more LoC, but improves latency & area

A Key-Value Store Cache



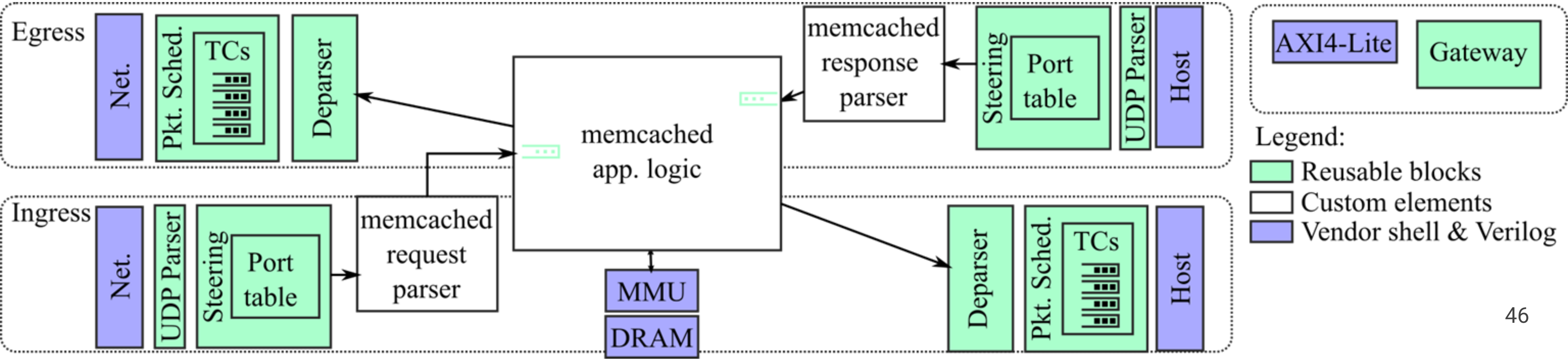
Eran, H., Zeno, L., Tork, M., Malka, G., & Silberstein, M. NICA: An infrastructure for inline acceleration of network applications. In 2019 USENIX Annual Technical Conference (USENIX ATC 19).

Key-value store cache

Processes 16-byte GET hits at 40.3 Mtps.

For 75% hit rate: 9× compared to CPU-only.

Uses: hash tables, header processing, scheduler, control plane, programmable FIFOs, ...



More information & related work

For more information

- Our paper: Design Patterns for Code Reuse in HLS Packet Processing Pipelines, FCCM ’19.
- Productive parallel programming for FPGA with HLS, Johannes de Fine Licht and Torsten Hoefler, ETH.
- Xilinx’s Introduction to FPGA Design with Vivado High-Level Synthesis and Vivado Design Suite User Guide – High-Level Synthesis

Related work

- https://github.com/Xilinx/HLS_packet_processing
  Xilinx (unofficial) packet processing library builds parsers with boost::mpl.
- Module-per-Object: a human-driven methodology for C++-based high-level synthesis design, Silva et al., FCCM ’19.

Conclusion

What does it take to write packet processing hardware in C++ HLS?

Try out `ntl`: https://github.com/acsl-technion/ntl

Thank you!

Questions?