18-643 Lecture 14: A Study in HLS and Streaming

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today: see how easy FPGAs are to use nowadays in a design study applying HLS to streaming data analytics
- Notices
  - Handout #6: Lab 3, due noon, 10/30 (or 11/3)
  - Handout #7: Paper Review, sign-up due 10/27
  - Midterm in class, Wed 10/25
  - Project proposal due 10/30!!
- Readings (see lecture schedule online)
  - *FPGA Optimization Guide for Intel® oneAPI Toolkits*
  - *FPGA for Aggregate Processing: The Good, The Bad, and The Ugly* [Eryilmaz, et al. 2021]

HLS Manifesto

1. On the right problems, HLS **CAN** produce high quality results

   Not all computation run faster in HW (HLS or RTL)

2. By the right designers, HLS **CAN** match RTL quality

   How one writes C code matters (HW or SW)

Performance must be in the program, for HLS to find it.

How hard is it to use FPGA in 2023?

- **No harder** than using GPGPUs through CUDA or OpenCL
- E.g., Intel DPC++/oneAPI
  - single-source heterogenous programming, as simple as,

    icpx –fsycl –fintelfpga . . . main.cpp

    ./a.out
  - functionally portable across systems with CPU/GPU/FPGA . . . etc.
  - **BUT**, getting good performance requires human hands

This is always the case, on any platform

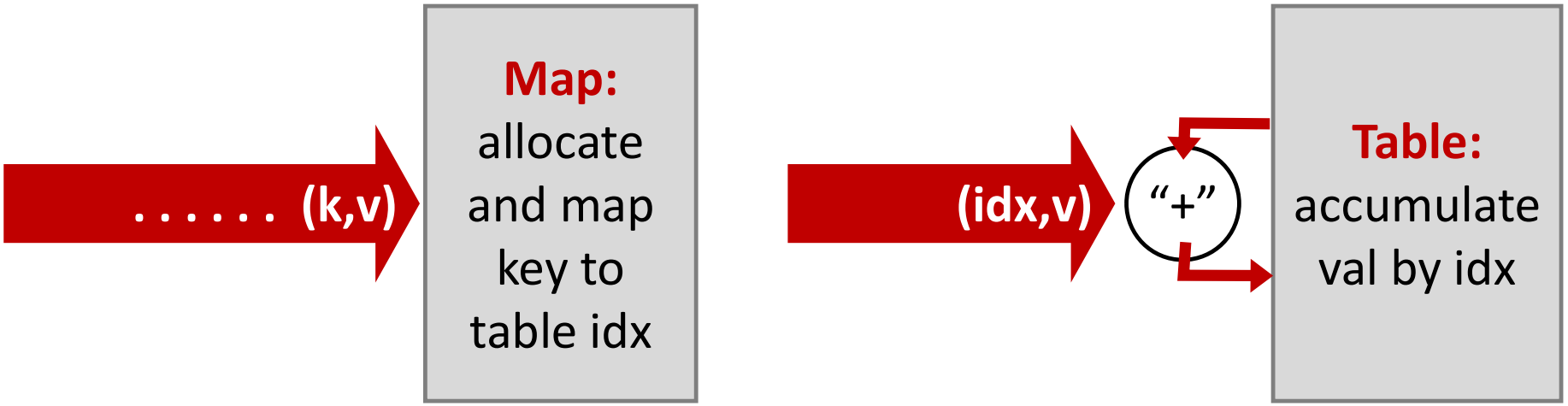
- If only I also could write Python and call performance libraries . . .

To Sharpen the Question

- How hard is to get **good performance** on FPGA?
  - for nuts-and-bolts **kernel developers**
  - for **application developers** using kernels
  - in which **application domain**?

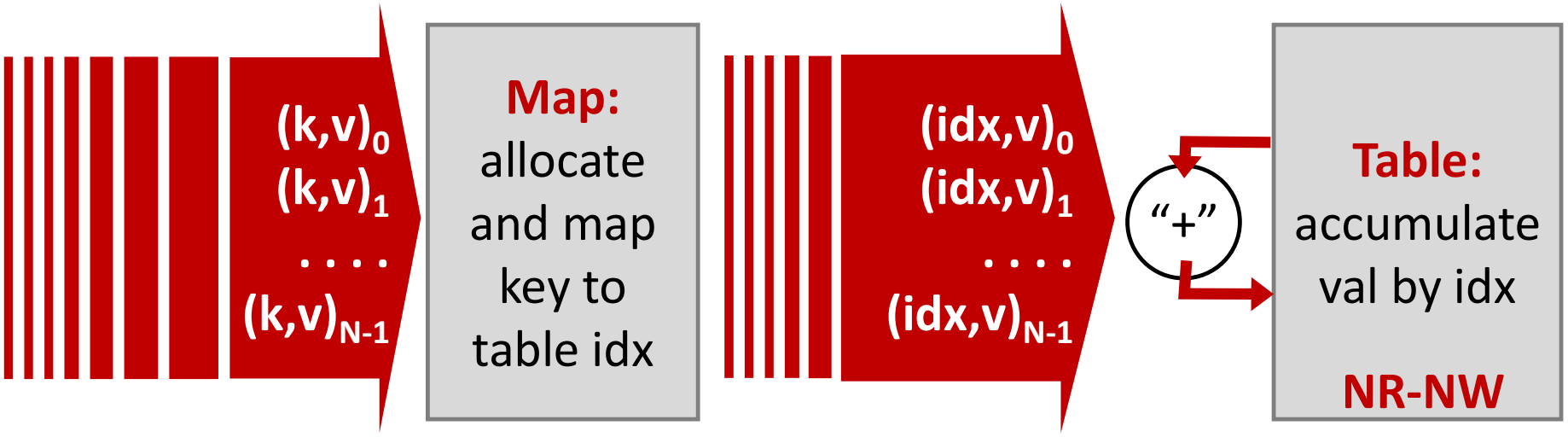
Design Example: Aggregation by Key

- Input: a stream of key-value pairs: **(k_0,v_0), (k_1,v_1), ... (k_{n-1},v_{n-1})**
- Report at the end:
  - distinct **keys** that appeared in stream
  - each key's aggregated **summary value** *(sum, min, average, etc.)*



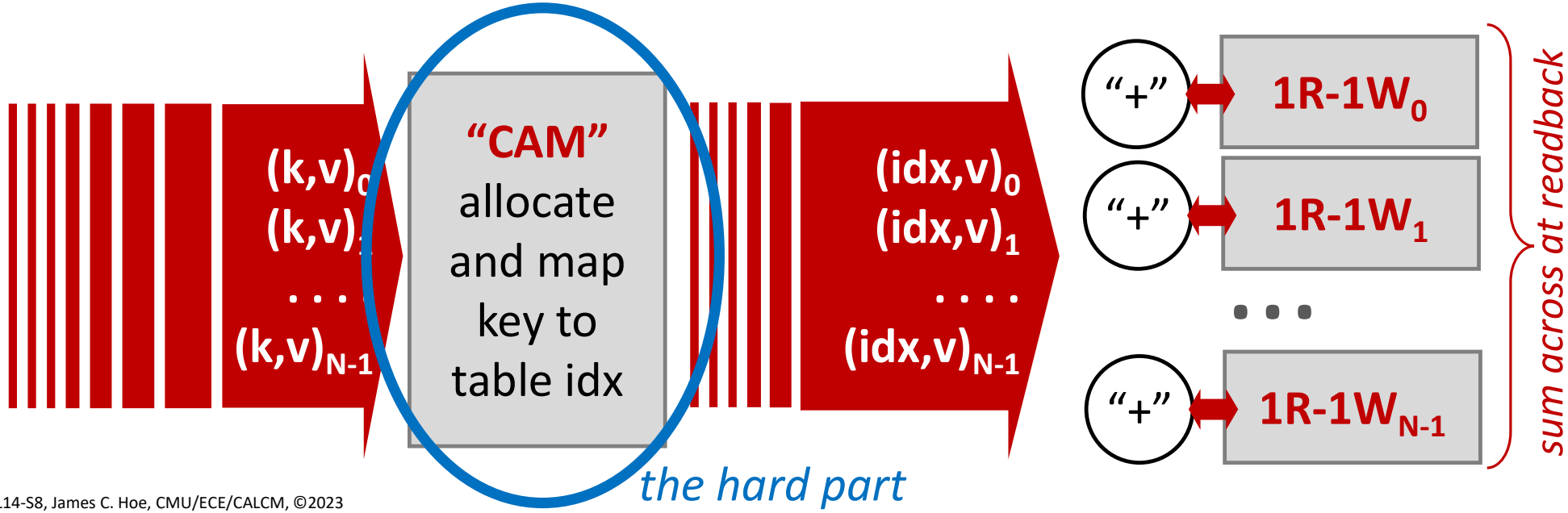
More Precisely

- Assume
  - 32-bit int **key** and **value**; accumulate by simple addition
  - no more than **G** distinct keys expected (commonly under **64**)
  - **N** key-value pairs per cycle throughput desired (e.g., **1~32**)



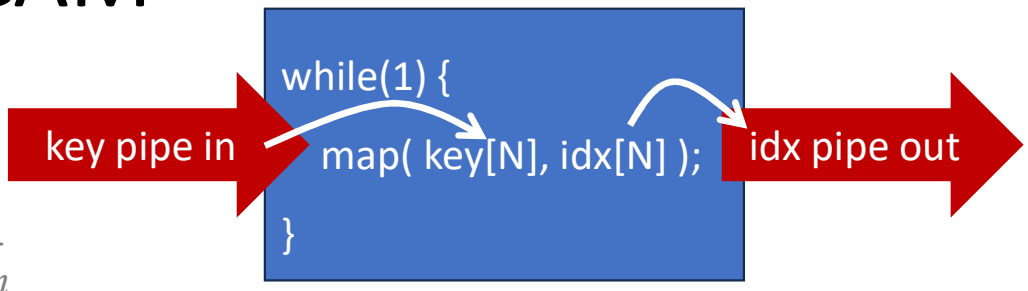
Still More Precisely

- Assume
  - 32-bit int **key** and **value**; accumulate by simple addition
  - no more than **G** distinct keys expected (commonly under **64**)
  - **N** key-value pairs per cycle throughput desired (e.g., **1~32**)



1st Try at describing a CAM

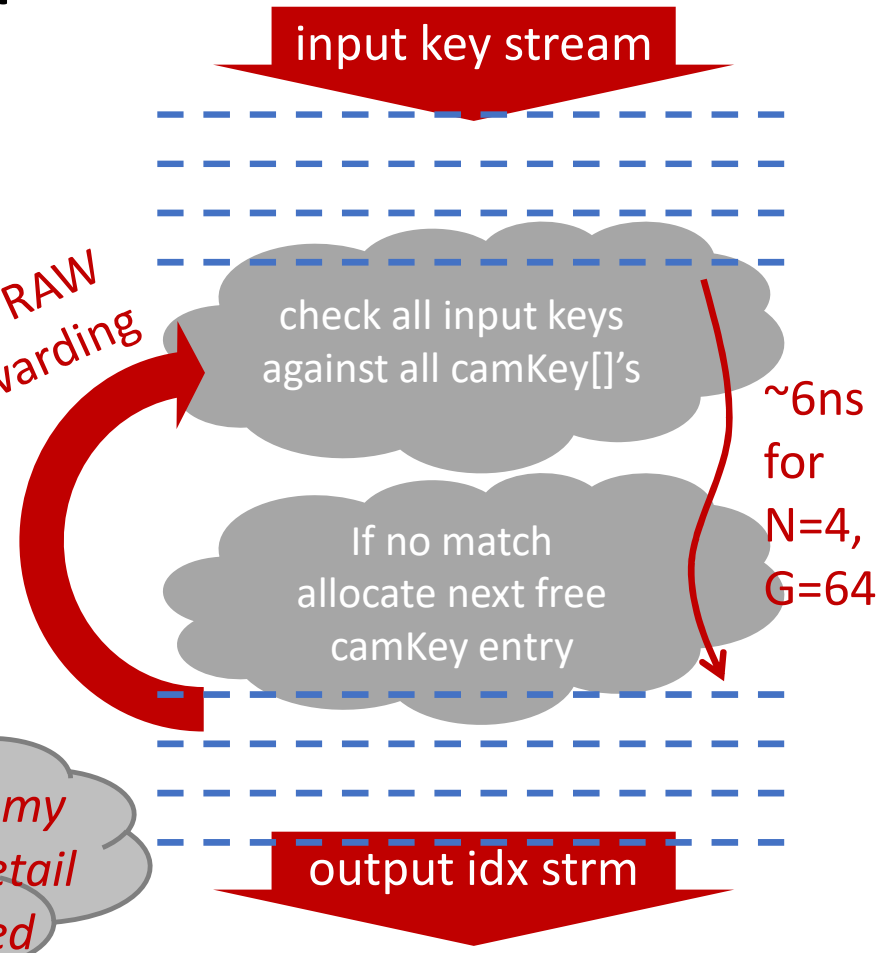
```
0  bool camValid [G] = {}; int camKey [G] = {};
   int nextFree = 0;

   void map(int key[N], int idx[N]) {    // no duplicate keys …
     bool hit[N] = {};                   //  … in same iteration
5
#pragma unroll
     for (int t = 0; t < N ; t++) {              // check all key[N]
#pragma unroll
       for (int g = 0; g < G ; g++) {   // against all camKey[G]
         if (camValid [g] && (key[t] == camKey [g])) {
           idx[t] = g;                           // mapping found
10         hit[t] =  true;  } } }

#pragma unroll
     for (int t = 0; t < N ; t++) {
       if (!hit[t]) {      // allocate free entry if key not mapped
         camValid[nextFree] = true;
15       camKey[nextFree] = key[t];
         idx[t] = nextFree;                      // new mapping
         nextFree++;   } } }
   }
```



- map(), called in a loop, becomes a pipeline of compiler-tuned depth
  - for-loops fully unrolled
  - **camKey[]** and **camValid[]** realized as registers for same-time access to all entries
  - new iter starts every cyc; forward state updates (**camValid** / **camKey**/ **nextFree**) from one iter to next

1st Try at describing a CAM

```
0  bool camValid [G] = {}; int camKey [G] = {};
   int nextFree = 0;

   void map(int key[N], int idx[N]) {    // no duplicate keys …
     bool hit[N] = {};                   //  … in same iteration
5
   #pragma unroll
     for (int t = 0; t < N ; t++) {              // check all key[N]
   #pragma unroll
       for (int g = 0; g < G ; g++) {   // against all camKey[G]
         if (camValid [g] && (key[t] == camKey [g])) {
           idx[t] = g;                           // mapping found
10         hit[t] =  true;  } } }

   #pragma unroll
     for (int t = 0; t < N ; t++) {
       if (!hit[t]) {      // allocate free entry if key not mapped
         camValid[nextFree] = true;
15       camKey[nextFree] = key[t];
         idx[t] = nextFree;
         nextFree++;   } } }
   }
```

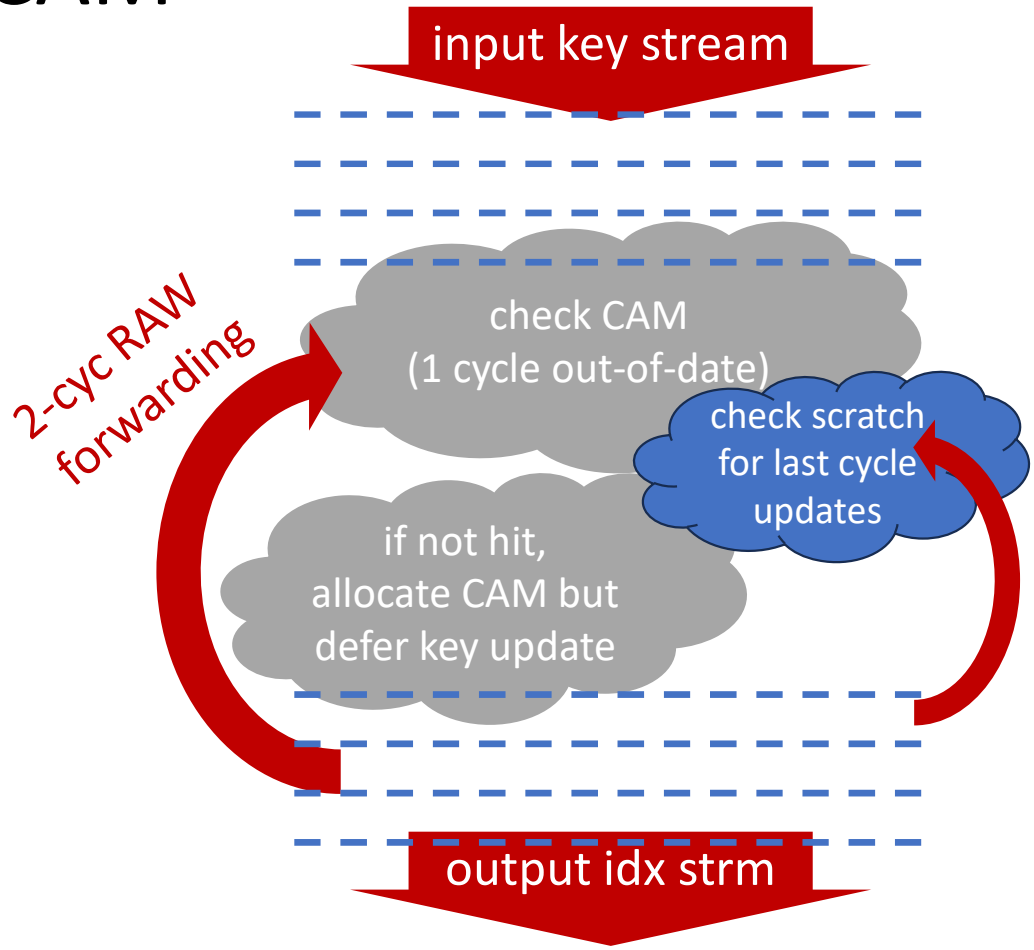


Started with picture in my head—only as much detail as shown—then coded

2nd Try at describing a CAM

- Large **G**-entry CAM is slow
  - better to update CAM one cycle later to cut critical path
  - but RAW hazard would stall pipeline every other cycle
- How about writings updates to an **N**-entry scratchpad
  - actual CAM writes can happen next iteration
  - next iteration reads check both CAM and scratchpad
  - new cyc time 3.7ns @ **N**=4, **G**=64

    ~ 1 gigapair/sec



2nd Try at describing a CAM

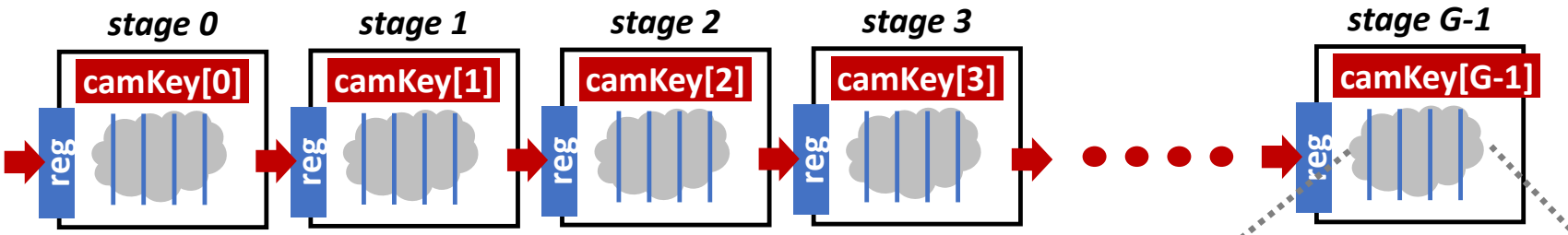

```
0  bool camValid [G] = {}; int camKey [G] = {};
   int nextFree = 0;
   // record deferred updates to next iteration
   int dfrValid[N] = {}; int dfrKey[N] = {}; int dfrIdx[N] = {};

5  void map(int key[N], int idx[N]) {      // no duplicate keys
     bool hit[N] = {};                     //    in same iteration
     // CAM lookup, same as before
#pragma unroll
     for (int t = 0; t < N ; t++) {
#pragma unroll
       for (int g = 0; g < G ; g++) {
10       if (camValid [g] &&(key[t] == camKey [g])) {
           idx[t] = g;                     // mapping found
           hit[t] =  true;  } } }

#pragma unroll
       for (int last = 0; last < N; last++) {   // check deferred
15       if (dfrValid[last] && (key[t] == dfrKey[last])) {
           idx[t]=dfrIdx[last];            // mapping found
           hit[t] = true;  } } }
```

```
20 // update CAM from deferred
   for (UIDX t = 0; t < N; t++) {
      if (dfrValid[t]) {
        camValid[dfrIdx[t]] = true;
        camKey[dfrIdx[t]] = dfrKey[t]; } }

25
   for (UIDX t = 0; t < N; t++) {
     if (!hit[t]) {
       // save deferred updates to CAM
       dfrValid[t] = true;
       dfrKey[t] = key[t];
30     dfrIdx[t] = nextFree;
       idx[t] = nextFree;
       nextFree++;
     } else {
       dfrValid[t] = false; } }
35 }
```

Again, started with the picture then coded

3rd Try at describing a CAM

- In streaming, latency does not matter
  - for each iteration, search **camKey[]** serially in **G** steps
  - overlap **G** iteration over **G** stages



- RAW feedback localize to individual **camKey[]**
  - new cyc time under 2ns on Agilex AGF014-2
  - $freq_{max}$ independent of **G** and **N**; cost=O(**N**^{2}+**GN**)
  - Terasic DE10 max out at 21 giga-pair/sec@**G**=64

if (**camKey[i]** is valid)
check for key match
else if (key still not matched)
allocate this **camKey[i]**

3rd Try at describing a CAM

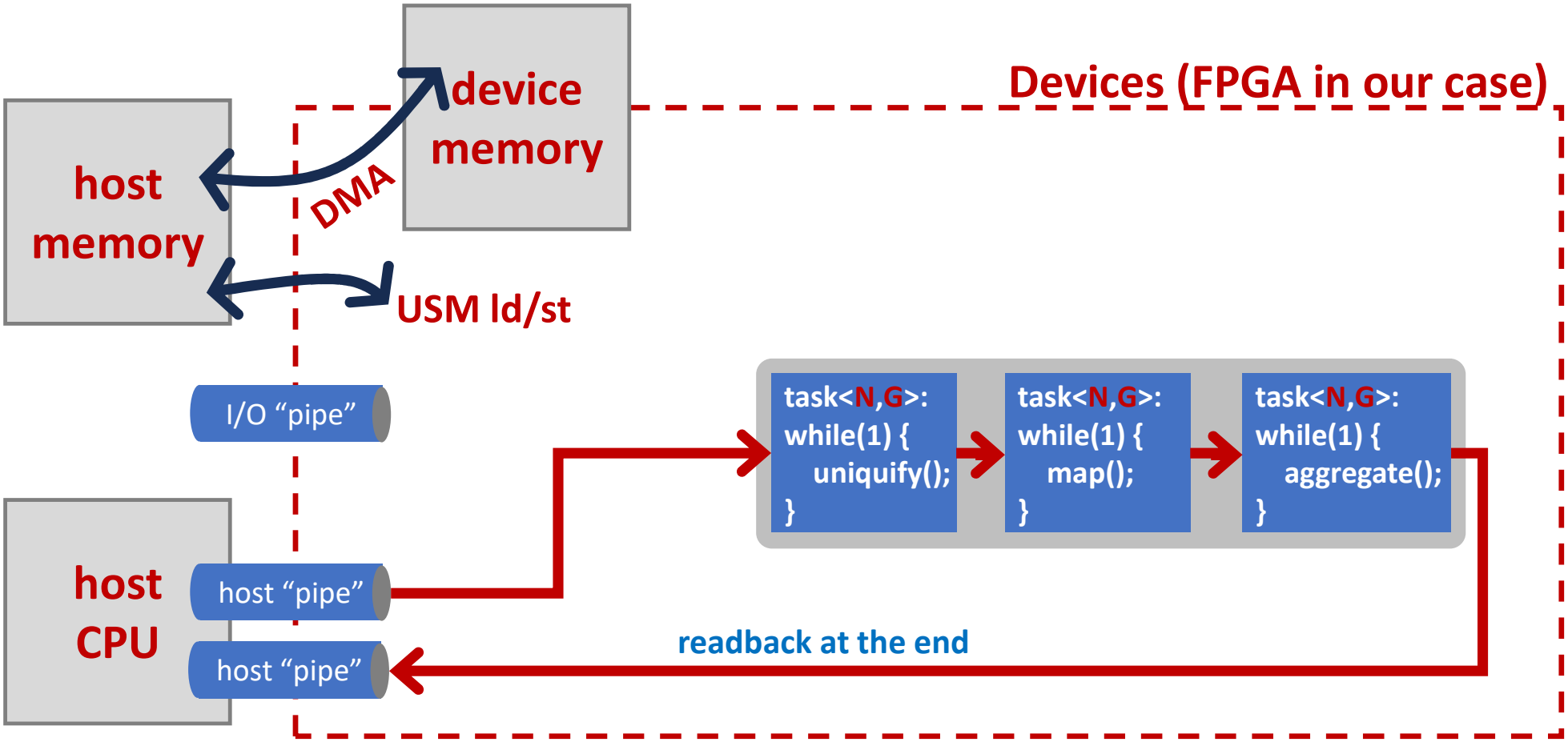

```
0  bool camValid [G] = {}; int camKey [G] = {};

   void map(int key[N], int idx[N]) {   // no duplicate keys . . .
     bool hit[N] = {};                  //      in same iteration

#pragma unroll
5  for (int g = 0; g < G; g++) {        // for each camKey stage
       if (camValid[g]) {               // is a valid camKey stage?
#pragma unroll
           for (int t = 0; t < N ; t++) {
               if (key[t] == camKey[g]) {
                   idx[t] = g; hit[t]=true              // matched
10     }}} else {                     // not yet allocated camKey stage
#pragma unroll
           for (int t = 0; t < N ; t++) {
               if (!hit[t]) {     // allocate to first unmapped key
                   camValid[g] = true;
                   camKey[g] = key[t];
15                 hit [t] = true;  idx [t] = g;
                   break; // rest continue on next stage } } }
     }
   }
```

- Compiler can turn this into the intended systolic pipeline

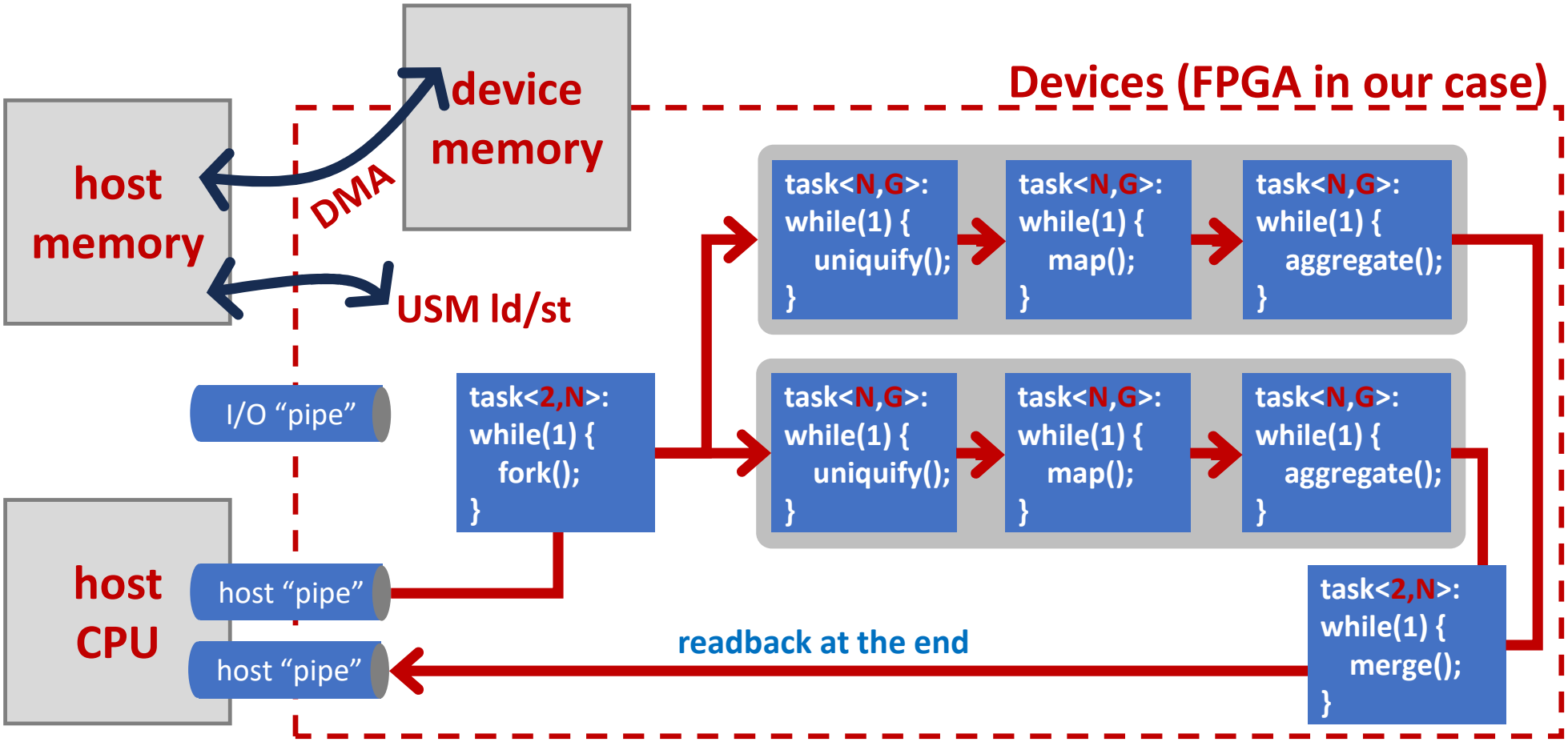
  More intricate than you think

- Compiler cannot transform **try-1** or **try-2** into **try-3**'s structure and timing
- Compiler did help me get here faster by making it not painful to try out ideas

main.cpp: a View into System and Application

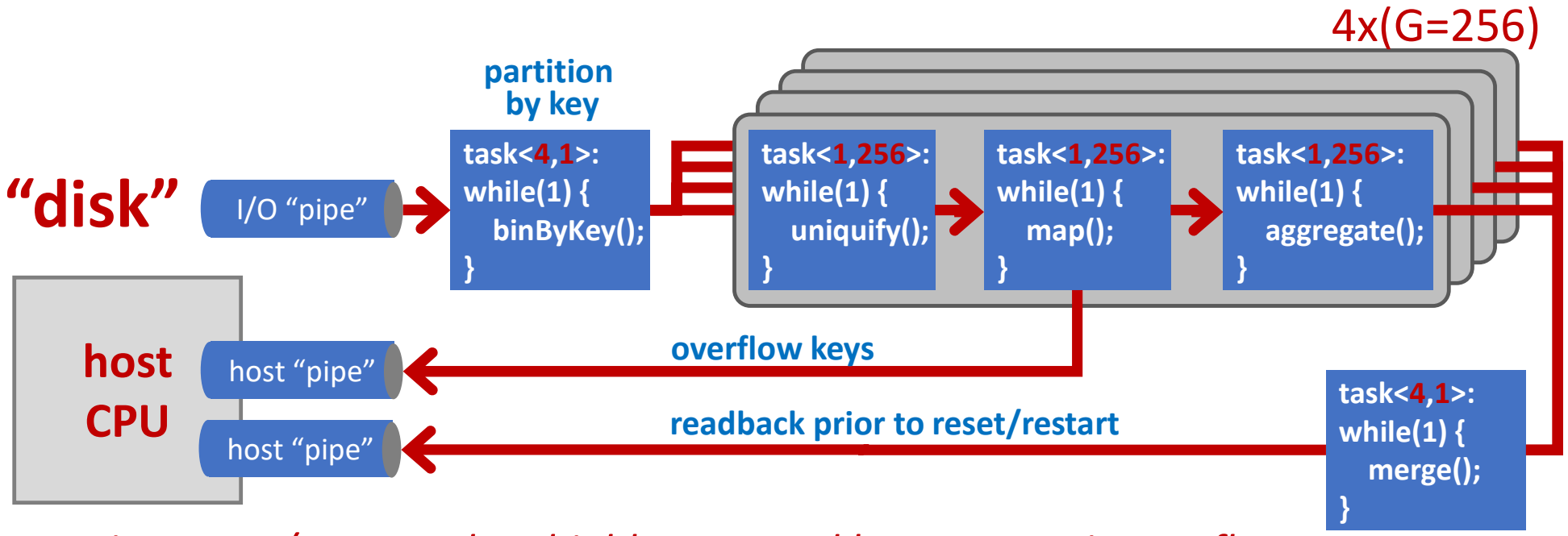


main.cpp: a View into System and Application



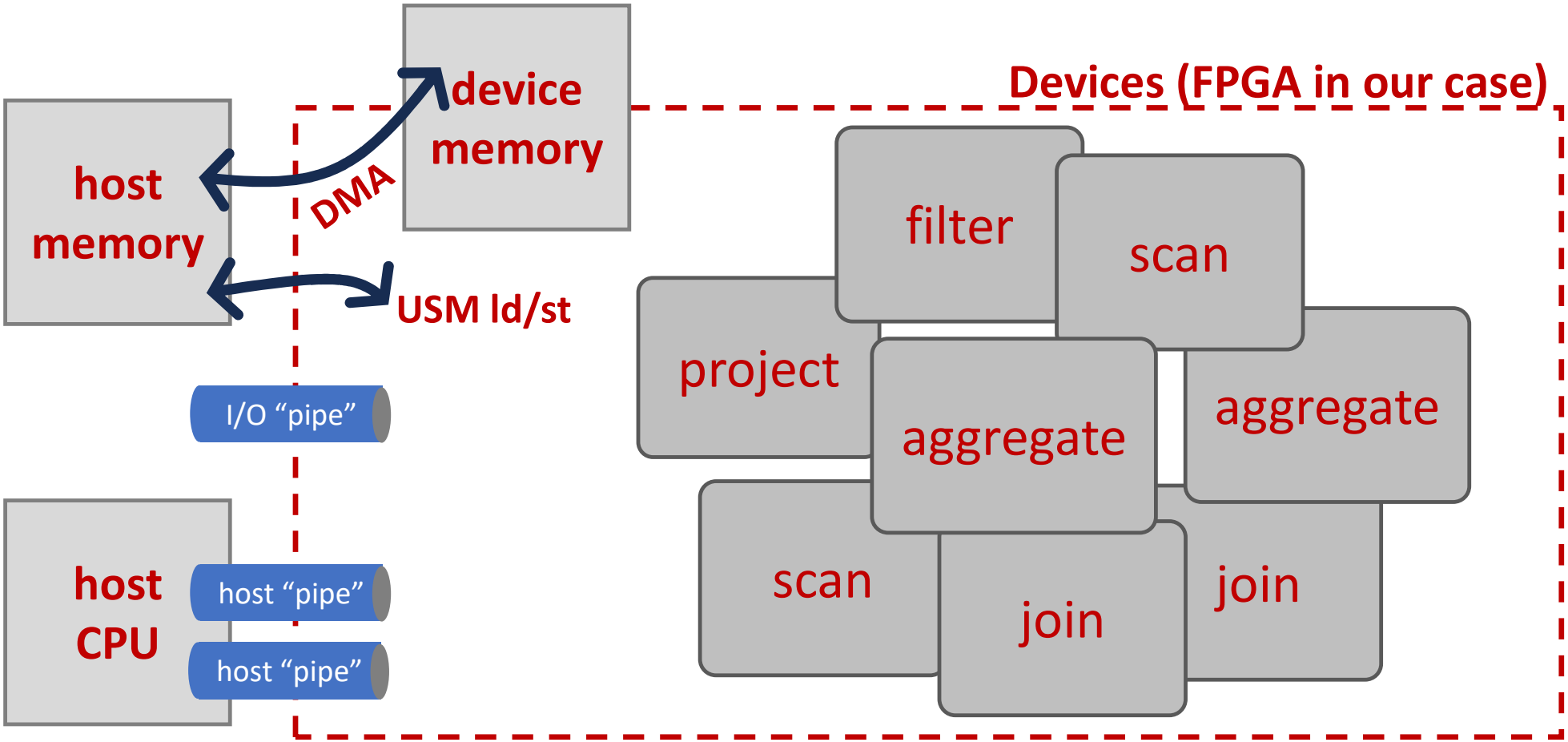
Suppose we want G=1M

- key-value stream is billions long
- keys have temporal locality, e.g., slow drifting "working set" < 512 keys



Host just reset/restart when highly-repeated keys appear in overflow

Performance Library for Data Query Processing



Recap So Far

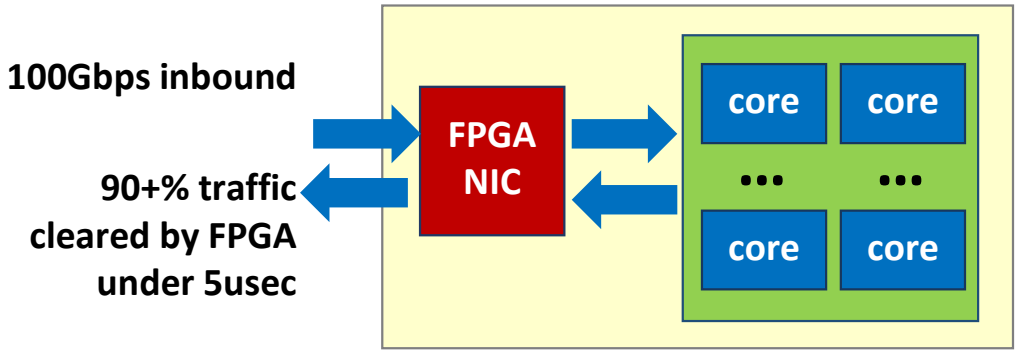
- Using DPC++/oneAPI for streaming aggregation
  - orders of magnitude easier than RTL (but performance never "easy")
  - no practical quality limitations (speed or cost)

    Even if you want to finalize in RTL, start from where I left off in SYCL
- Maintainability and reusability of IPs
  - conciseness of code, powerful parameterization *(thanks to HLS)*
  - standard interfaces *(thanks to pipes)*
  - plug-and-play modularity *(thanks to "software engineering")*
- Supporting application developers with library
  - they can read the kernel code (even if they can't write it)
  - trivial to customize value type or aggregation function
  - edit main.cpp to build new data-analytic pipeline from kernels??

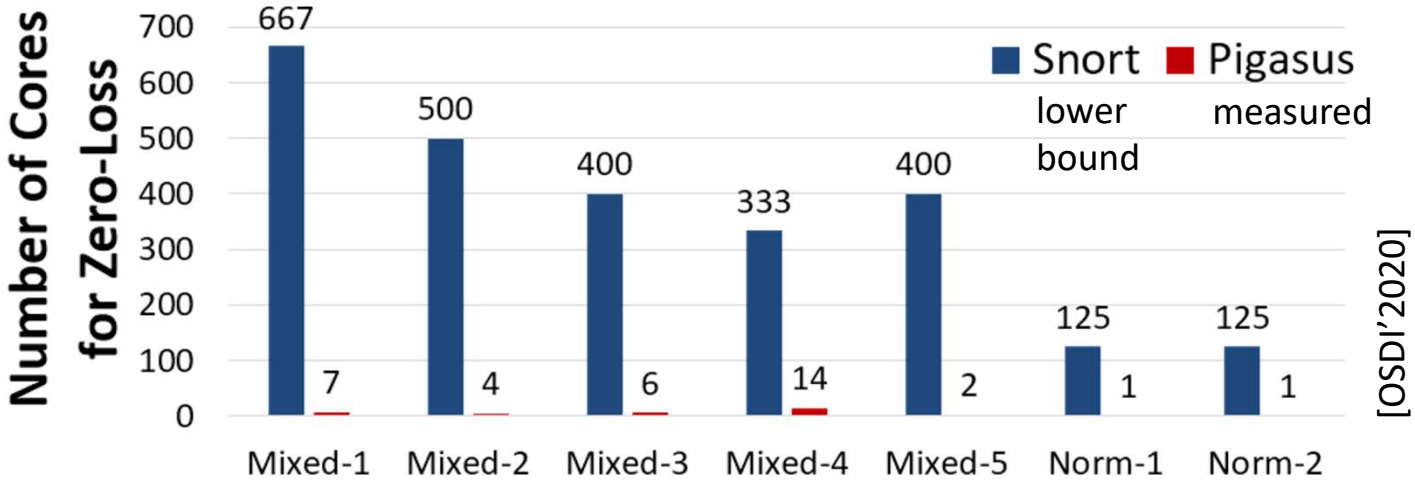
What is FPGA good for anyways?

Stream data processing is one answer

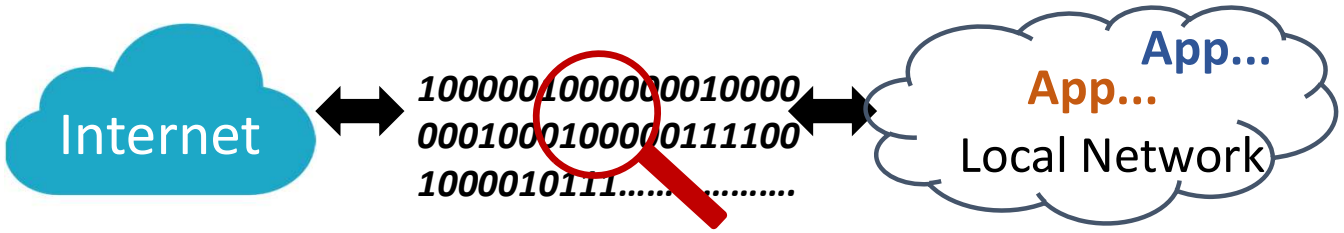
The Pigasus Saga: Deep Packet Inspection at 100Gbps using 1 FPGA NIC + 1 CPU



Checks every packet payload byte against SNORT registered ruleset



A Hard Problem for CPU and GPU



- Check packet payload against a set (10s K) of elaborate rules (e.g., string matching and regular expressions)

Fine-grained, irregular parallelism over byte stream

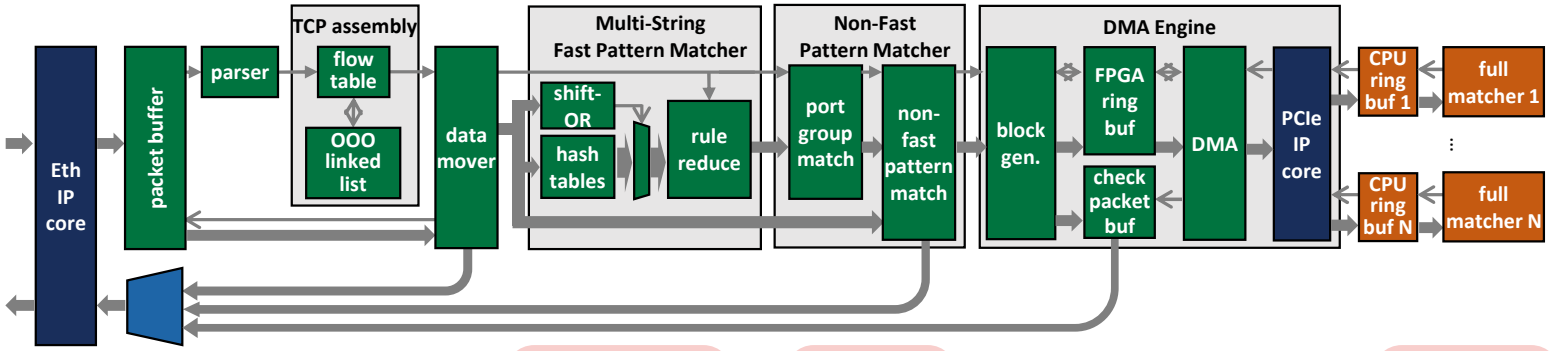
- Performed inline with traffic

Must keep up with line rate

- Stop malicious packet from propagating

Latency matters

Found a nice solution using FPGA



Ethernet *100 Gbps*

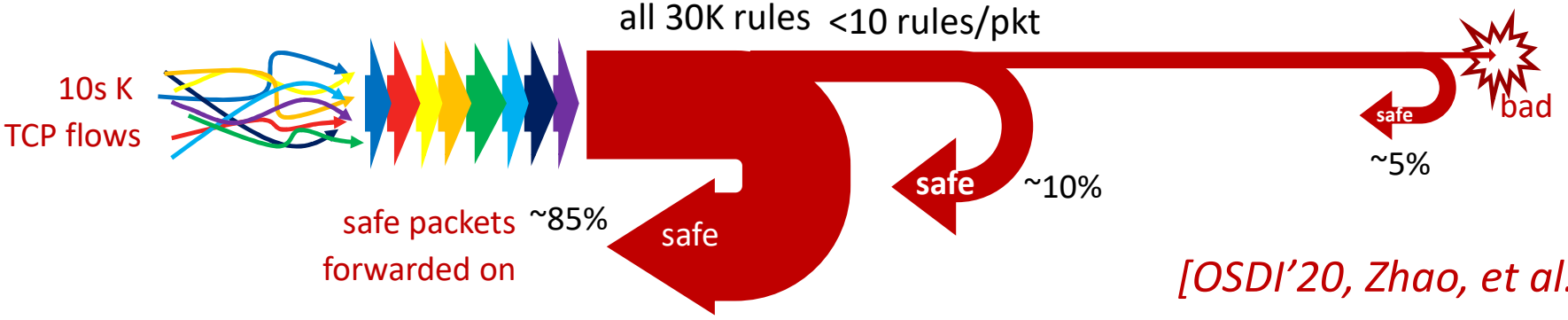
TCP flow reassembly *100 Gbps* 100k flows

"fast pattern" matching *100 Gbps* all 30K rules

"non-fast pattern" matching *~15 Gbps* <10 rules/pkt

offloading to CPU *~5 Gbps*

regex matching (on CPU) <5 per pkt



[OSDI'20, Zhao, et al.]

Pigasus Opensource Experience

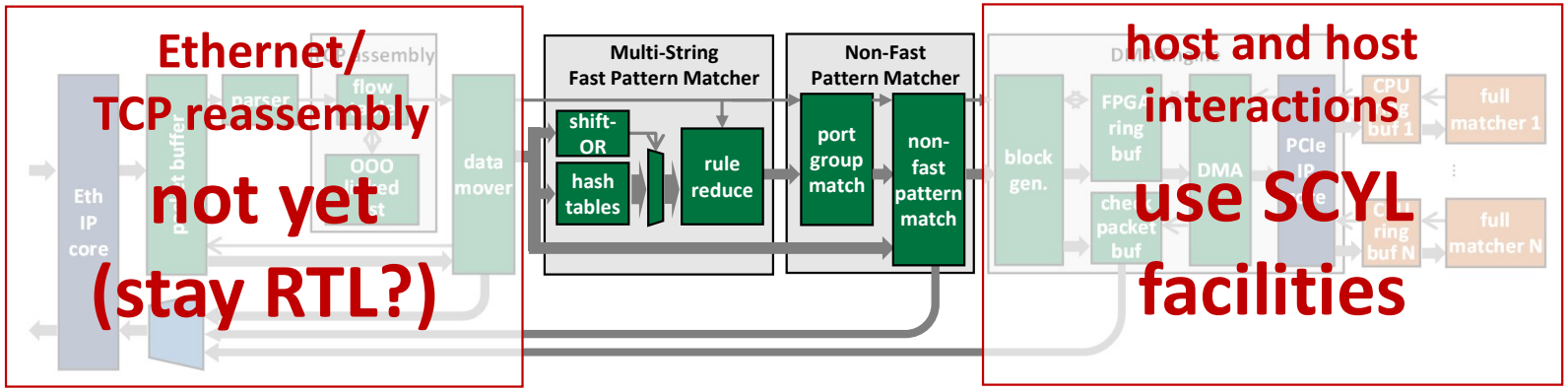
https://github.com/crossroadsfpga/pigasus

- Opensourced entire RTL code base in 2020
  - many downloads, several recreated "as is", couple Xilinx porting attempts
  - so far no known continuing work by anyone or ourselves

- What did the project die?
  - 80k lines of SystemVerilog code
  - too hard to understand and too fragile to modify
  - requires a high-level of combined algorithm and RTL design expertise
- It ***IS*** well engineered (parameterized, generated, interfaced, etc.)
  - Zhipeng created many derivative designs to study in thesis
  - code base effectively abandoned when Zhipeng graduated

Pigasus Rebooted in SYCL HLS

- Status:



- Completed DPI stages—same speed and cost as RTL
- You can understand Pigasus HLS code if you can understand Hyperscan's source code
- Useful IPs separable from Pigasus toward a **Data Analytic Library**
  - multistring (10s K) pattern matching: any string anywhere in stream
  - multistring signature check: packet contains all strings in signature
  - a variety of common utilities

Parting Thoughts

- FPGAs hold tremendous promise in stream data processing (transformation, inspection, and analytics)
- If FPGAs were easier to use, we have a "killer app"
- Applications people have to want to work with FPGAs
  - think Python, not RTL or CUDA/OpenCL/SYCL
  - deliver ease and performance through good libraries
- Code base must be maintainable and reusable for efforts to grow
  - moving to high-level design is inevitable
  - plant "software engineering" into HW language, tool, designer mindset

DPC++/oneAPI is very, very close to being an answer!!