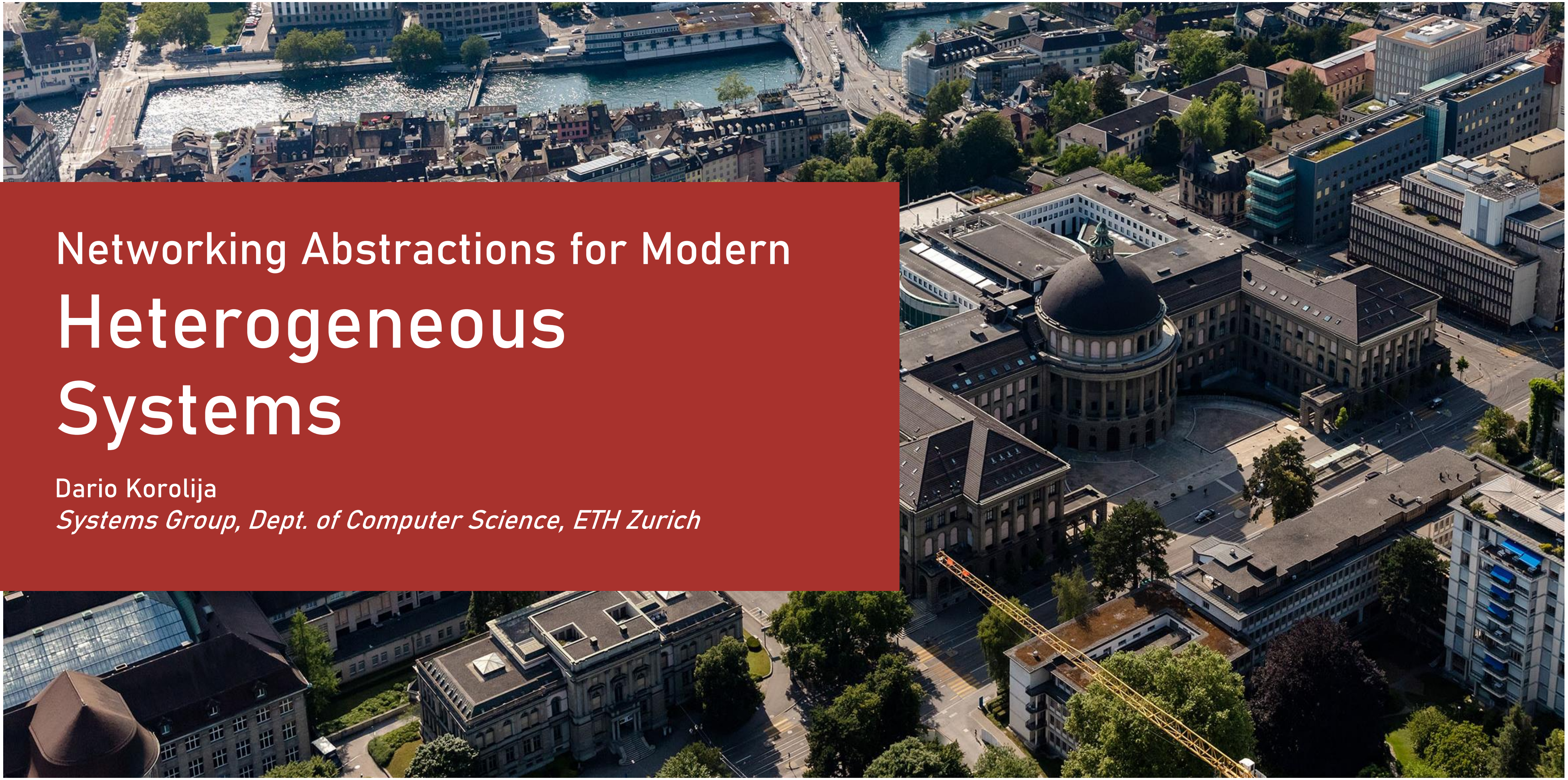


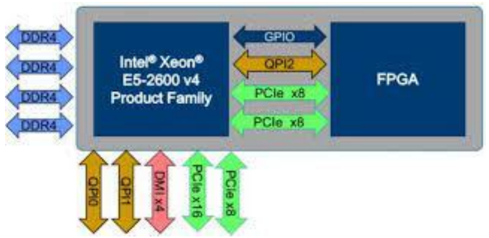
Networking Abstractions for Modern Heterogeneous Systems

Dario Korolija

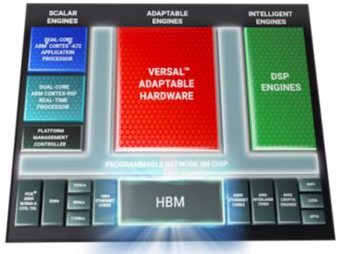
Systems Group, Dept. of Computer Science, ETH Zurich

Introduction...

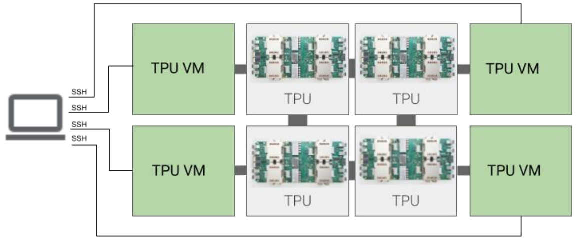
- Scpecialized hardware becoming a reality
  - *Amazon, Microsoft, Google, Alibaba, Intel, AMD ...*



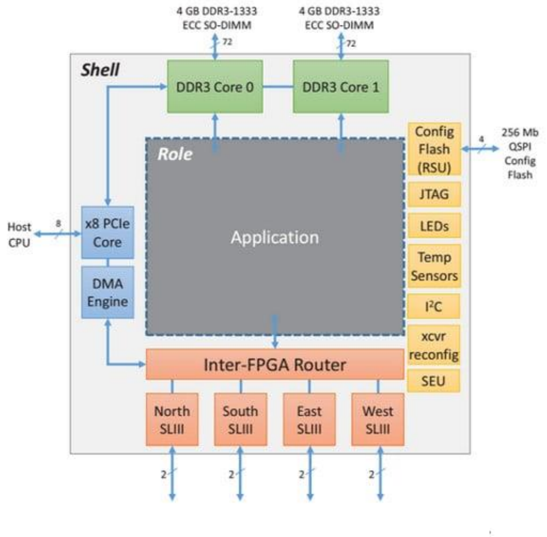
Intel HARP



AMD Versal



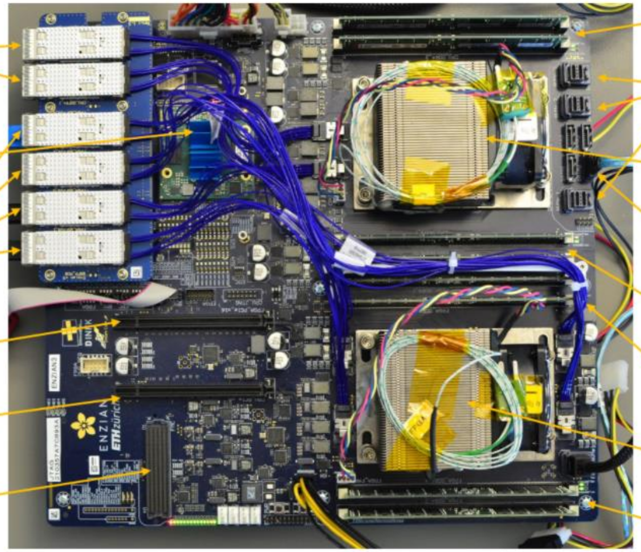
Google TPU



Microsoft Catapult

- One built within System group at ETH:

ENZIAN

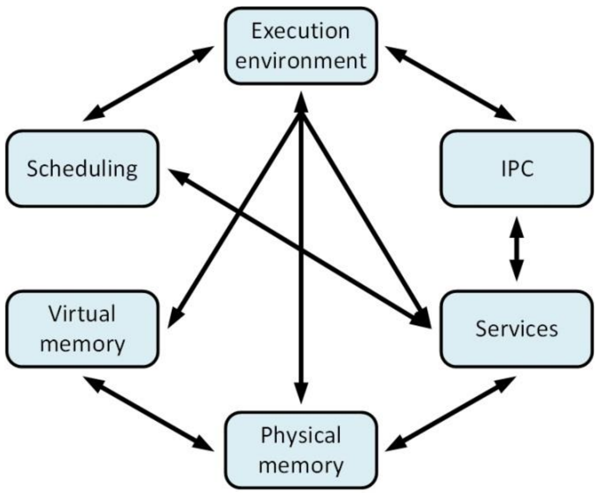


Microkernel for FPGAs

Hybrid computing system

- Plenty of research, focused on individual functionalities only
- Coyote provides a complete minimal core set of essential features on which further services can be used

Interdependence



A Hypervisor for Shared-Memory FPGA Platforms

Jiacheng Ma[*] Gefei Zuo[*] Kevin Loughlin[*] Xiaohe Cheng[§] Yanqiang Liu[†]
Abel Mulugeta Eneyew[‡] Zhengwei Qi[†] Baris Kasikci[*]
[*]University of Michigan [§]Hong Kong University of Science and Technology
[†]Shanghai Jiao Tong University [‡]Addis Ababa Institute of Technology

Sharing, Protection, and Compatibility for Reconfigurable Fabric with AMORPHOS

Ahmed Khawaja[1], Joshua Landgraf[1], Rohith Prakash[1],
Michael Wei[2], Eric Schkufza[2], Christopher J. Rossbach[3]
[1]*The University of Texas at Austin* [2]*VMware Research Group*
[3]*The University of Texas at Austin and VMware Research Group*

The Feniks FPGA Operating System for Cloud Computing

Jiansong Zhang[§] Yongqiang Xiong[§] Ningyi Xu[§] Ran Shu[§†] Bojie Li[§‡]
Peng Cheng[§] Guo Chen[§] Thomas Moscibroda[§]
[§] Microsoft Research [†]Tsinghua University [‡]USTC
{jiazhang,yqx,ningyixu,v-ranshu,v-bojli,pengc,guoche,moscitho}@microsoft.com

A Cloud-Scale Acceleration Architecture

Adrian M. Caulfield Eric S. Chung Andrew Putnam
Hari Angepat Jeremy Fowers Michael Haselman Stephen Heil Matt Humphrey
Puneet Kaur Joo-Young Kim Daniel Lo Todd Massengill Kalin Ovtcharov
Michael Papamichael Lisa Woods Sitaram Lanka Derek Chiou Doug Burger

Microsoft Corporation

Abstract—**Hyperscale datacenter providers have struggled to balance the growing need for specialized hardware (efficiency) with the economic benefits of homogeneity (manageability). In this paper we propose a new cloud architecture that uses reconfigurable logic to accelerate both network plane functions and applications. This Configurable Cloud architecture places a layer of reconfigurable logic (FPGAs) between the network switches and the servers, enabling network flows to be programmably transformed at line rate, enabling acceleration of local applications running on the server, and enabling the FPGAs to communicate directly, at datacenter scale, to harvest remote FPGAs unused by their local servers. We deployed this design over a production server bed, and show how it can be used for both service acceleration (Web search ranking) and network acceleration (encryption of data in transit at high-speeds). This architecture is much more scalable than prior work which used secondary rack-scale networks for inter-FPGA communication. By coupling to the network plane, direct FPGA-to-FPGA messages can be achieved at comparable latency to previous work, without the secondary network. Additionally, the scale of direct inter-FPGA messaging is much larger. The average round-trip latencies observed in our measurements among 24, 1000, and 250,000 machines are under 3, 9, and 20 microseconds, respectively. The Configurable Cloud architecture has been deployed at hyperscale in Microsoft's production datacenters worldwide.**

I. Introduction

Modern hyperscale datacenters have made huge strides with improvements in networking, virtualization, energy efficiency, and infrastructure management, but still have the same basic structure as they have for years: individual servers with multicore CPUs, DRAM, and local storage, connected by the NIC through Ethernet switches to other servers. At hyperscale (hundreds of thousands to millions of servers), there are significant benefits to maximizing homogeneity; workloads can be migrated fungibly across the infrastructure, and management is simplified, reducing costs and configuration errors.

Both the slowdown in CPU scaling and the ending of Moore's Law have resulted in a growing need for hardware specialization to increase performance and efficiency. However, placing specialized accelerators in a subset of a hyperscale infrastructure's servers reduces the highly desirable homogeneity. The question is mostly one of economics: whether it is cost-effective to deploy an accelerator in every new server, whether it is better to specialize a subset of an infrastructure's new servers and maintain an ever-growing number of configurations, or whether it is most cost-effective to do neither. Any specialized accelerator must be compatible with the target workloads through its deployment lifetime (e.g. six years: two years to design and deploy the accelerator and four years of server deployment lifetime). This requirement is a challenge given both the diversity of cloud workloads and the rapid rate at which they change (weekly or monthly). It is thus highly desirable that accelerators incorporated into hyperscale servers be programmable, the two most common examples being FPGAs and GPUs.

Both GPUs and FPGAs have been deployed in datacenter infrastructure at reasonable scale without direct connectivity between accelerators [1], [2], [3]. Our recent publication described a medium-scale FPGA deployment in a production datacenter to accelerate Bing web search ranking using multiple directly-connected accelerators [4]. That design consisted of a rack-scale fabric of 48 FPGAs connected by a secondary network. While effective at accelerating search ranking, our first architecture had several significant limitations:

- The secondary network (a 6x8 torus) required expensive and complex cabling, and required awareness of the physical location of machines.
- Failure handling of the torus required complex re-routing of traffic to neighboring nodes, causing both performance loss and isolation of nodes under certain failure patterns.
- The number of FPGAs that could communicate directly, without going through software, was limited to a single rack (i.e. 48 nodes).
- The fabric was a limited-scale "bolt on" accelerator, which could accelerate applications but offered little for enhancing the datacenter infrastructure, such as networking and storage flows.

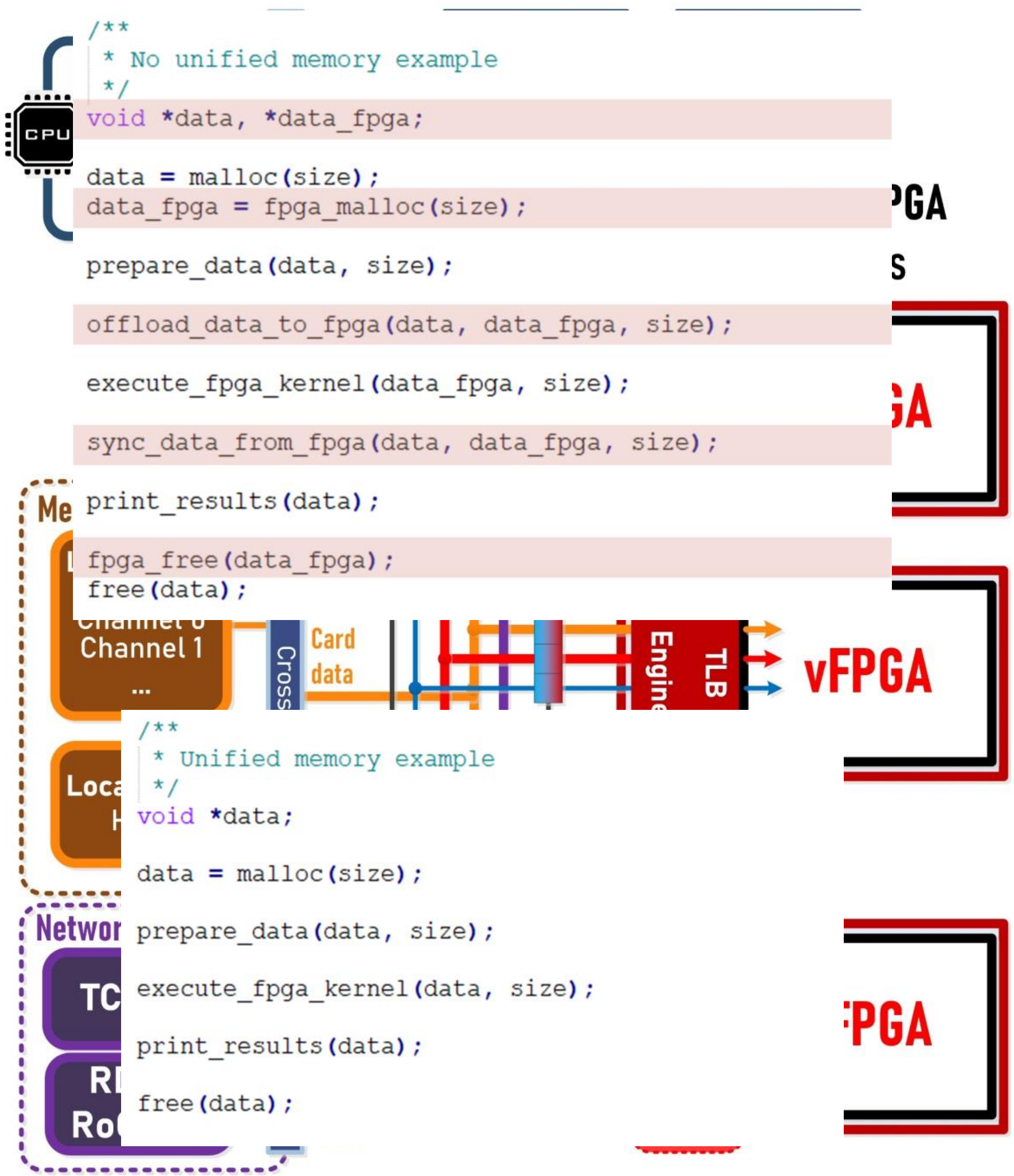
In this paper, we describe a new cloud-scale, FPGA-based acceleration architecture, which we call the *Configurable Cloud*, which eliminates all of the limitations listed above with a single design. This architecture has been — and is being — deployed in the majority of new servers in Microsoft's production datacenters across more than 15 countries and 5 continents. A Configurable Cloud allows the datapath of cloud communication to be accelerated with programmable hardware. This datapath can include networking flows, storage flows, security operations, and distributed (multi-FPGA) applications.

The key difference over previous work is that the accelera-

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

Coyote
System Architecture

- ❖ Microkernel for FPGAs
  - ➢ Multiple isolated untrusted vFPGAs
  - ➢ Spatial and temporal sharing (PR)
  - ➢ Dynamic reconfiguration (scheduler)
  - ➢ Virtualized memory
  - ➢ Unified memory (HMM)
  - ➢ HBM and DRAM striping service
  - ➢ Shared TCP/IP service
  - ➢ Shared RDMA service
  - ➢ Unified logic interface -> portability
  - ➢ RTL and HLS support
  - ➢ Runs on u50, u200, u250, u280, u55c, vcu118, *Enzian*



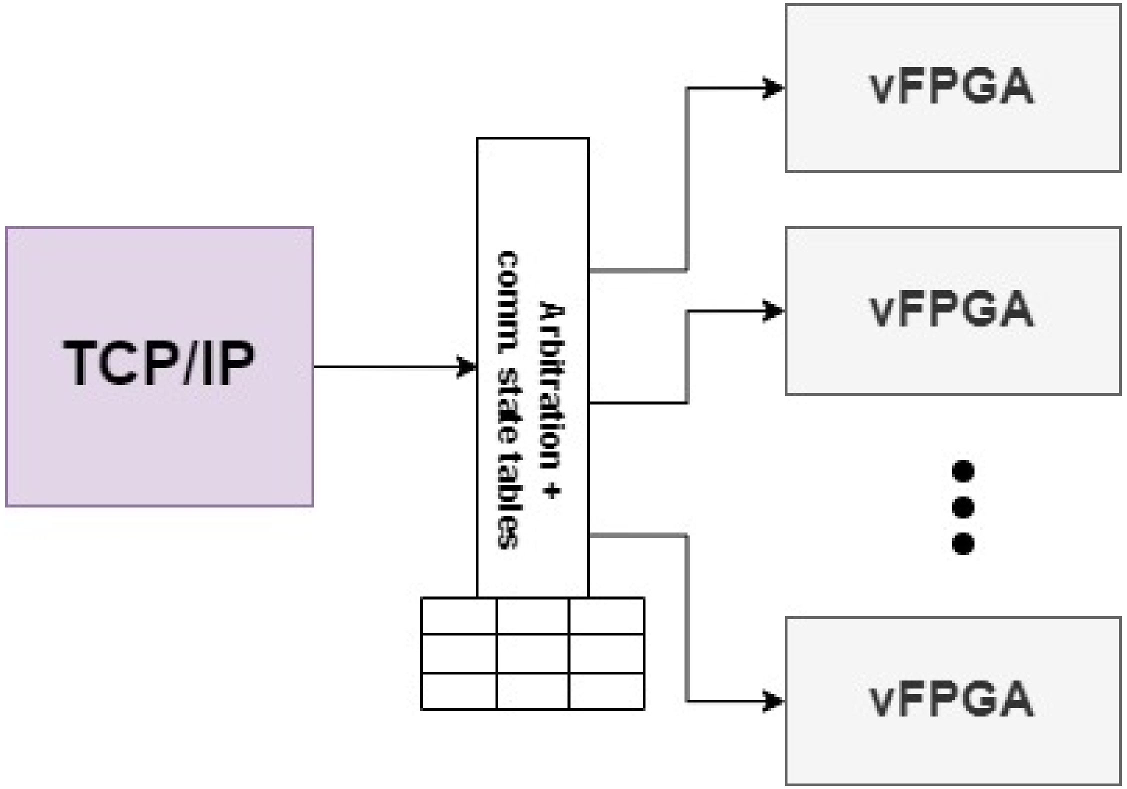
```
/**
 * No unified memory example
 */
void *data, *data_fpga;

data = malloc(size);
data_fpga = fpga_malloc(size);

prepare_data(data, size);

offload_data_to_fpga(data, data_fpga, size);

execute_fpga_kernel(data_fpga, size);

sync_data_from_fpga(data, data_fpga, size);

print_results(data);

fpga_free(data_fpga);
free(data);
```

```
/**
 * Unified memory example
 */
void *data;

data = malloc(size);

prepare_data(data, size);

execute_fpga_kernel(data, size);

print_results(data);

free(data);
```

Coyote

TCP/IP stack

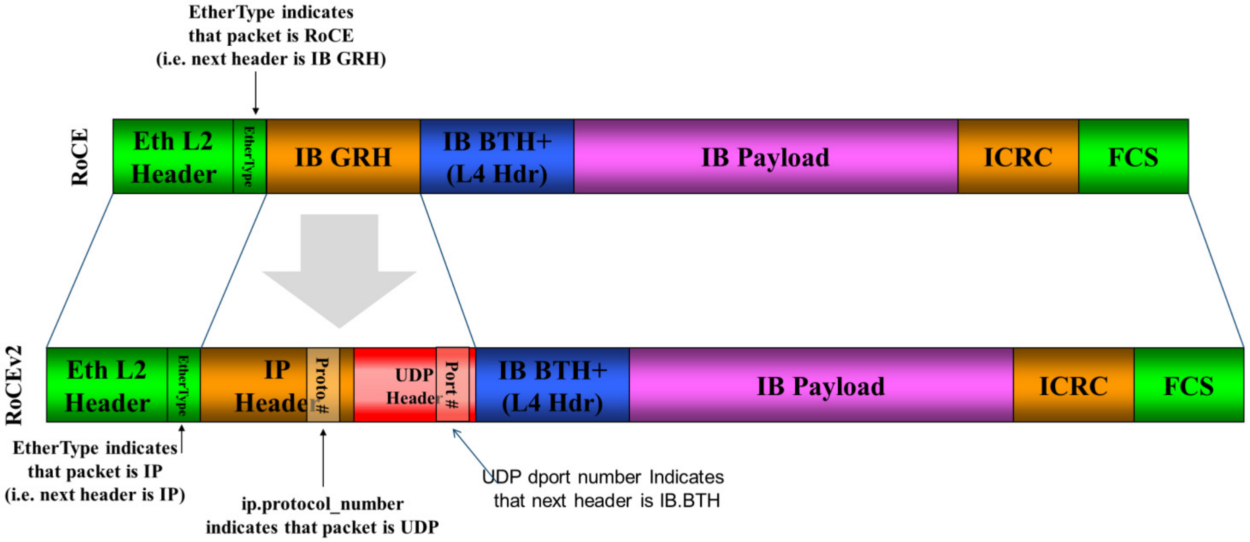
- Open source TCP/IP stack
- Added support for shared functionality between all applications running within an FPGA



Coyote

RDMA stack

- Open source RDMA stack (UC, RC)
- RDMA over Converged Ethernet (RoCE v2)
- Implemented on top of UDP/IPv4/IPv6 (far lower overhead than iWARP)
- InfiniBand (IB) transport packets over Ethernet (READ, WRITE, SEND)

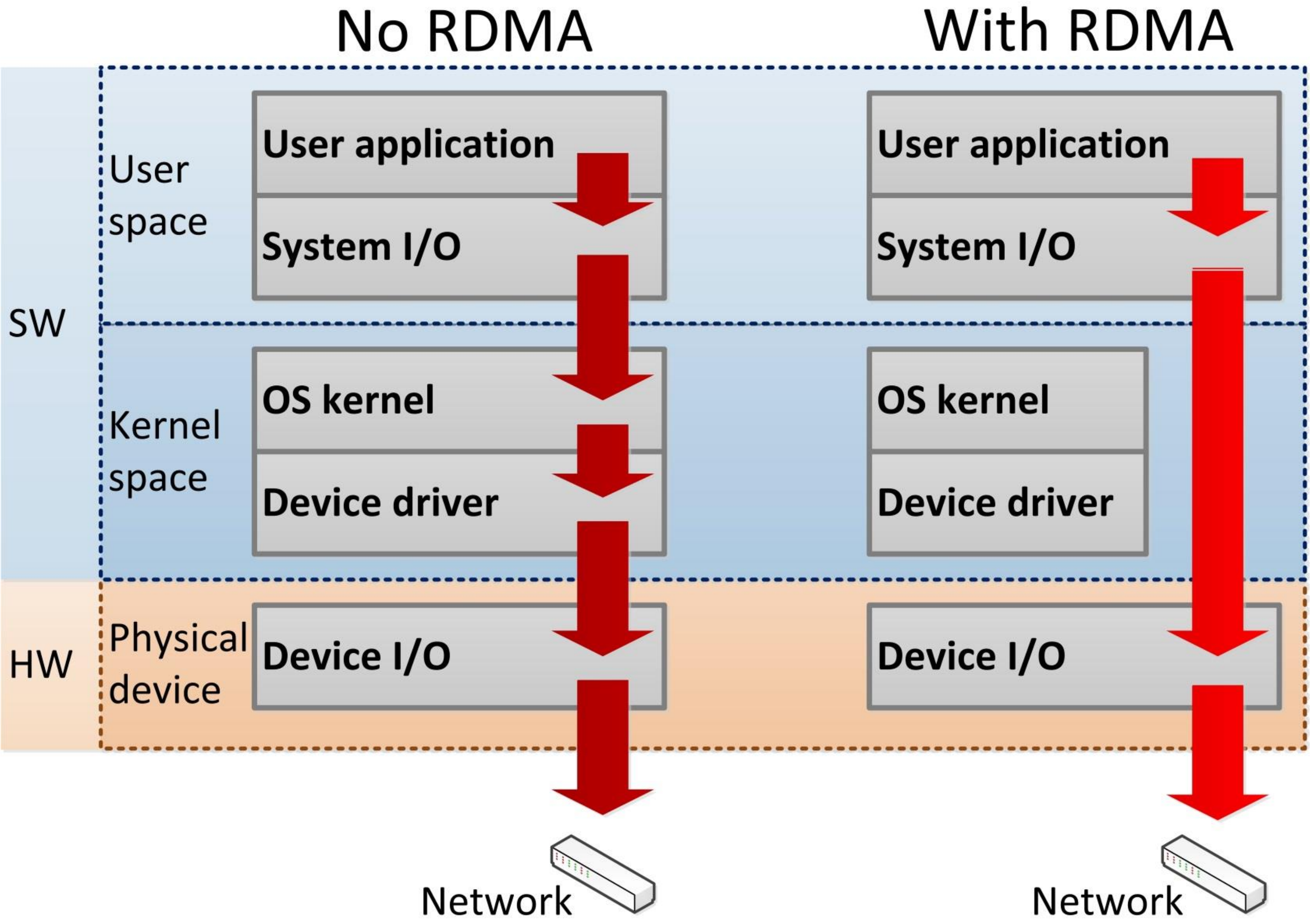

EtherType indicates that packet is RoCE (i.e. next header is IB GRH)
RoCE
Eth L2 Header
EtherType
IB GRH
IB BTH+ (L4 Hdr)
IB Payload
ICRC
FCS
RoCEv2
Eth L2 Header
EtherType
IP Header
Proto #
UDP Header
Port #
IB BTH+ (L4 Hdr)
IB Payload
ICRC
FCS
EtherType indicates that packet is IP (i.e. next header is IP)
ip.protocol_number indicates that packet is UDP
UDP dport number Indicates that next header is IB.BTH

https://docs.nvidia.com/networking/display/WINOFv55053000/RoCEv2

Coyote

RDMA advantages

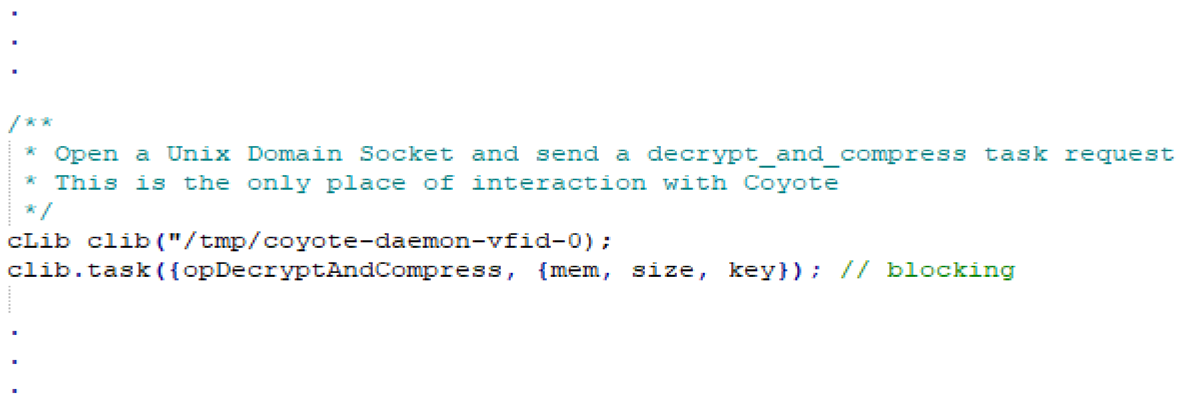
- Bypasses kernel space
- Zero-copy data movement
- Cheap pipelined processing (directly on the NIC)



Coyote

SW

- Layered parallelization potential
- User space abstractions
  - cSched – Coyote scheduler, reconfiguration controller
  - cProc – Coyote process, multiple can run within a single *vFPGA*
  - cThread – Coyote thread, multiple can run within a single *cProc*. Task level parallelism
  - cTask – Coyote task, arbitrary user variadic function executed by *cThreads*
  - cService - Coyote library daemon, background service, UDS for IPC



```
.
.
.
/**
 * Open a Unix Domain Socket and send a decrypt_and_compress task request
 * This is the only place of interaction with Coyote
 */
cLib clib("/tmp/coyote-daemon-vfid-0);
clib.task({opDecryptAndCompress, {mem, size, key}); // blocking
.
.
.
```

Coyote

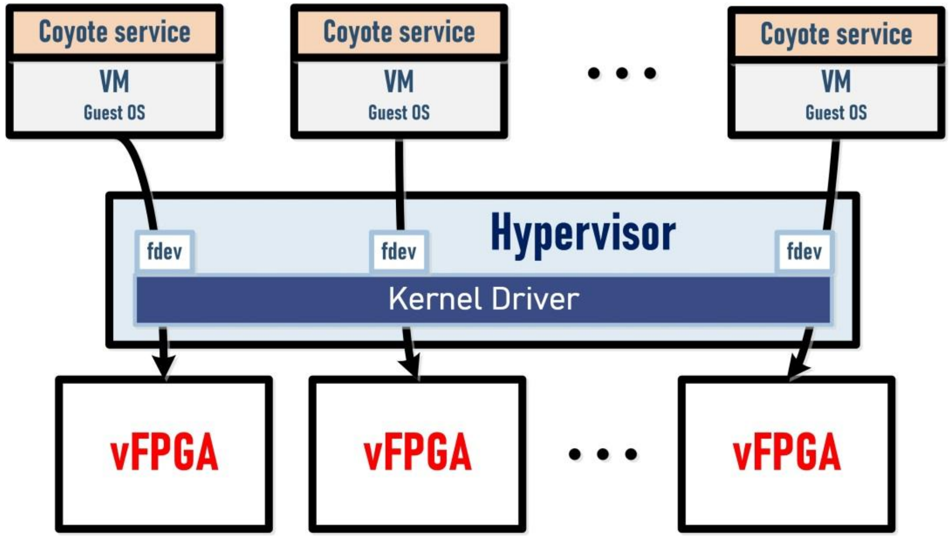
Repository

- Github: https://github.com/fpgasystems/Coyote.git
- Internal and external users
- Example designs (perf. runs, rdma, tcp/ip, rpc, hbm, dram, services)
- Documentation …

Coyote

Current Work

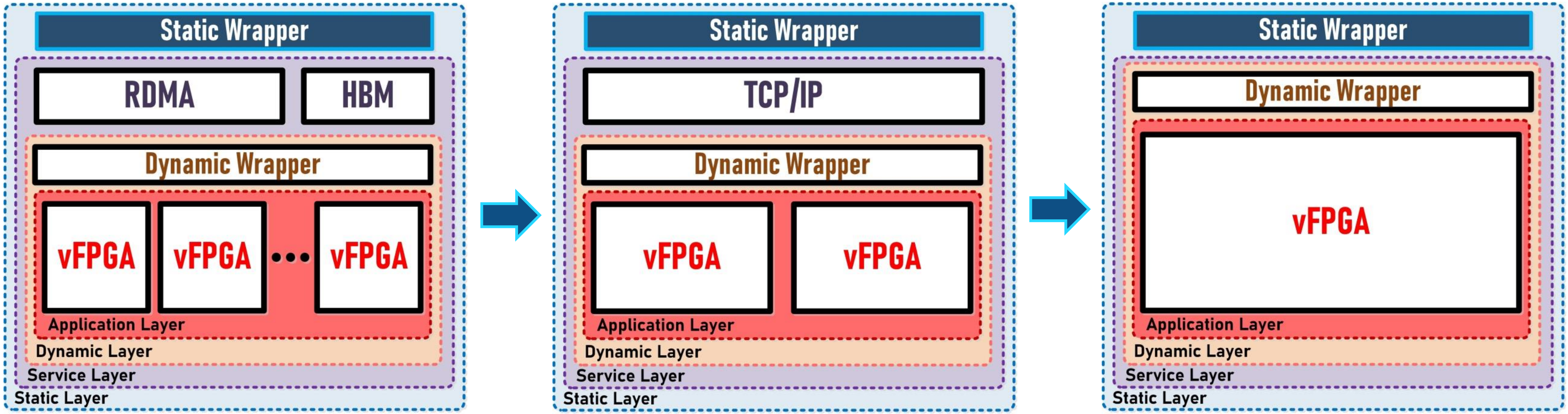
- Virtualization – can we pull vFPGAs all the way to VMs layer?
  - Virtual Function I/O (VFIO) Mediated devices



Coyote

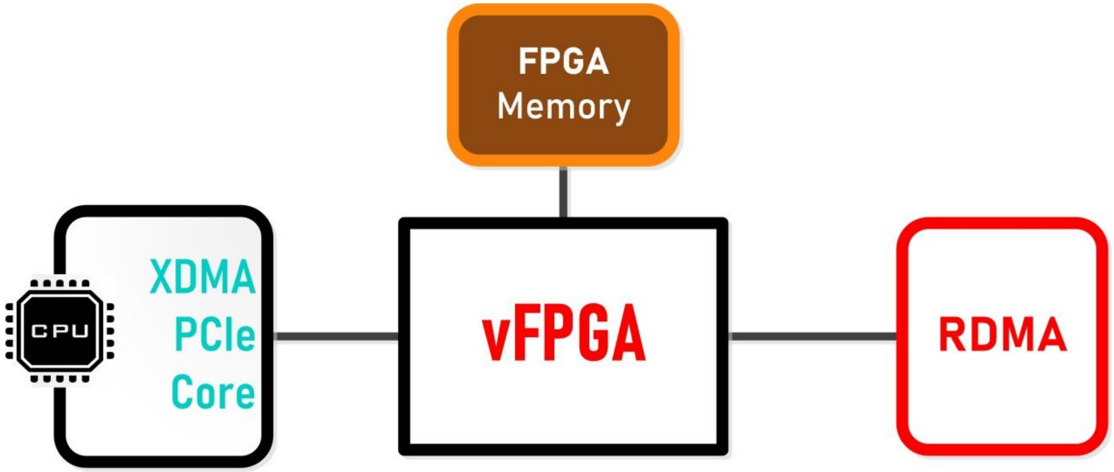
Current work

- Nested reconfiguration (*Nested-DFX*)
  - Static layer
  - Service layer
  - Dynamic layer
  - Application layer



Advantages of the system

- Networking advantages:
  - Streaming interfaces:
  - Kernel invocation overhead XRT~50us, Coyote: ~1-1.5us
  - RDMA



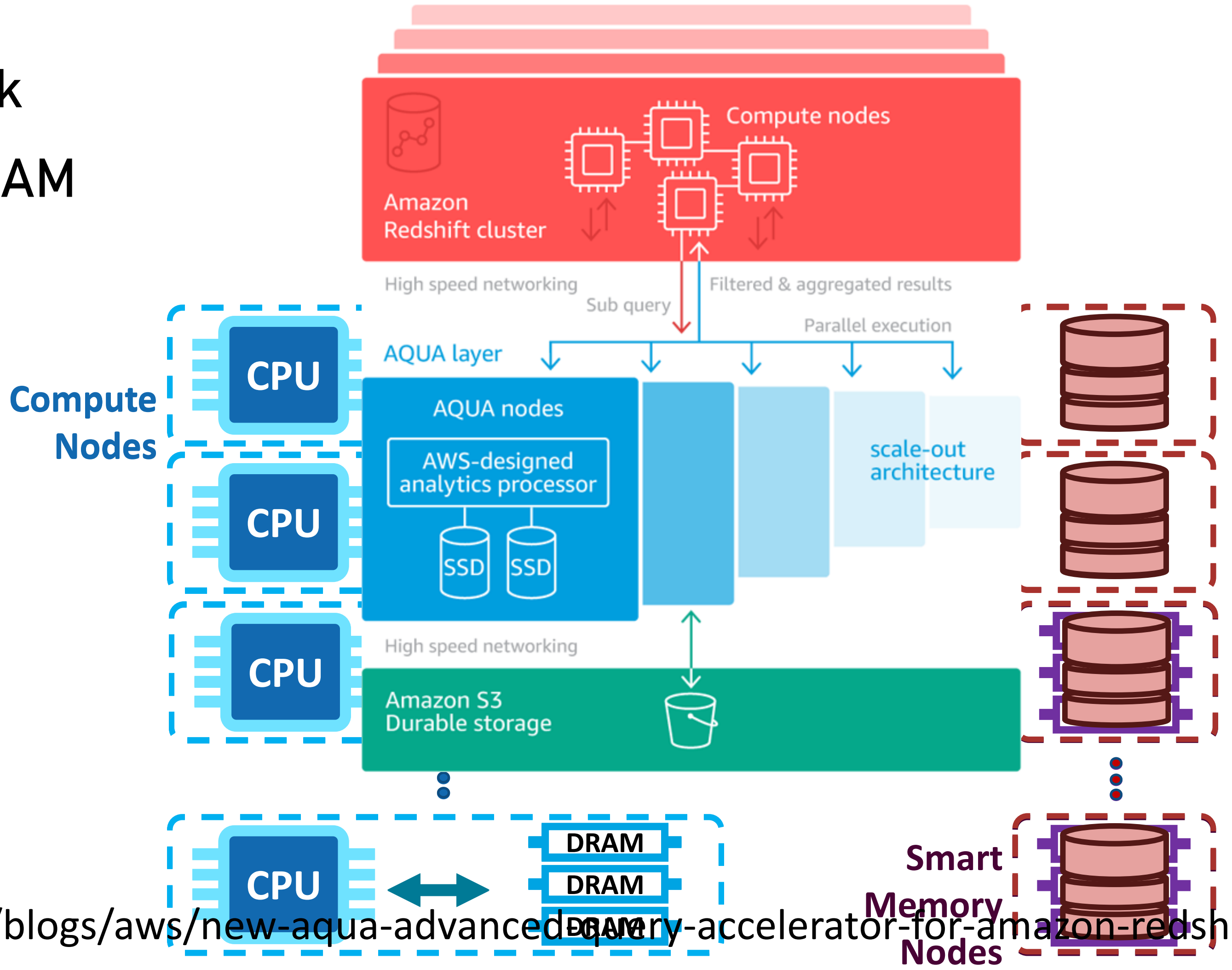
Farview

Motivation

- ❖ I/O overheads – main bottleneck
- ❖ More and more data in local DRAM
- ❖ Excessive data movement
- ❖ Memory capacity limitations

Disaggregation of
compute and storage

of compute and memory and storage

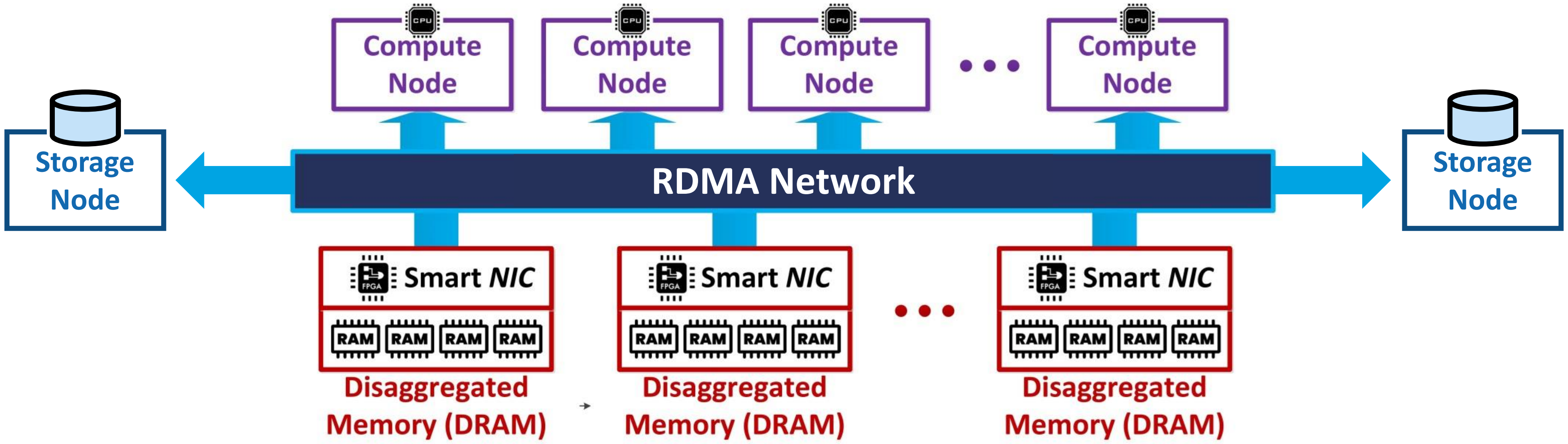


https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/

Farview

Introduction

- FPGA-based smart NIC making DRAM available as a pool of network attached memory accessible on demand over high performance RDMA network.
- Performs line-rate query processing with minimal overheads
- Farview is a disaggregated buffer cache with operator pushdown capabilities



Farview

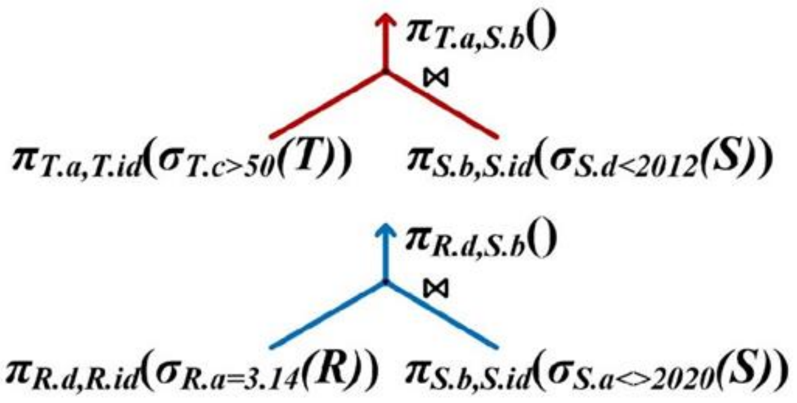
Overview

- Farview addresses inefficient data movement and memory capacity limitations
- Consider the following queries:

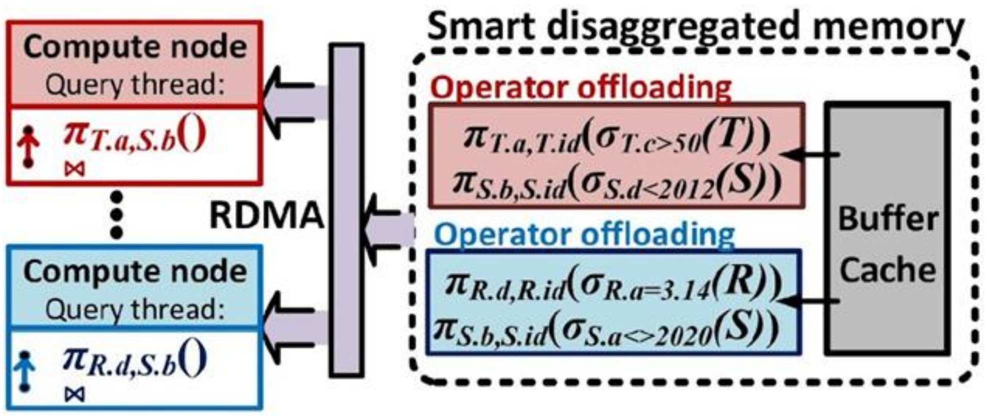
```
SELECT T.a, S.b
FROM T, S
WHERE T.id = S.id
AND T.c > 50 AND S.d < 2012;
```

```
SELECT R.d, S.b
FROM R, S
WHERE R.id = S.id
AND R.a = 3.14 AND S.a <> 2020;
```

- Farview centralizes the buffer cache and performs operator pushdown



$\pi_{T.a,S.b}()$
$\bowtie$
$\pi_{T.a,T.id}(\sigma_{T.c>50}(T))$
$\pi_{S.b,S.id}(\sigma_{S.d<2012}(S))$
$\pi_{R.d,S.b}()$
$\bowtie$
$\pi_{R.d,R.id}(\sigma_{R.a=3.14}(R))$
$\pi_{S.b,S.id}(\sigma_{S.a<>2020}(S))$

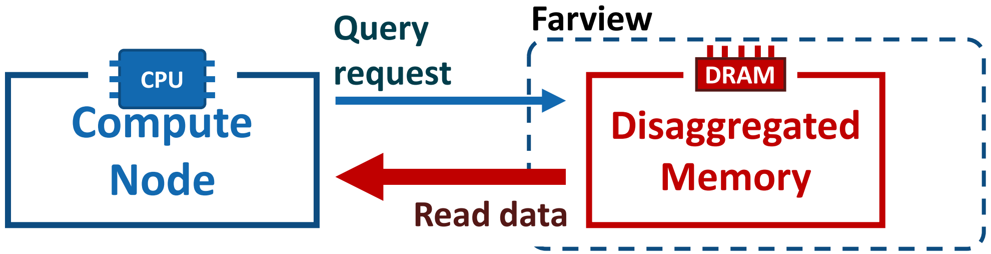


Compute node
Query thread:
$\pi_{T.a,S.b}()$
$\bowtie$
RDMA
Compute node
Query thread:
$\pi_{R.d,S.b}()$
$\bowtie$
Smart disaggregated memory
Operator offloading
$\pi_{T.a,T.id}(\sigma_{T.c>50}(T))$
$\pi_{S.b,S.id}(\sigma_{S.d<2012}(S))$
Operator offloading
$\pi_{R.d,R.id}(\sigma_{R.a=3.14}(R))$
$\pi_{S.b,S.id}(\sigma_{S.a<>2020}(S))$
Buffer Cache

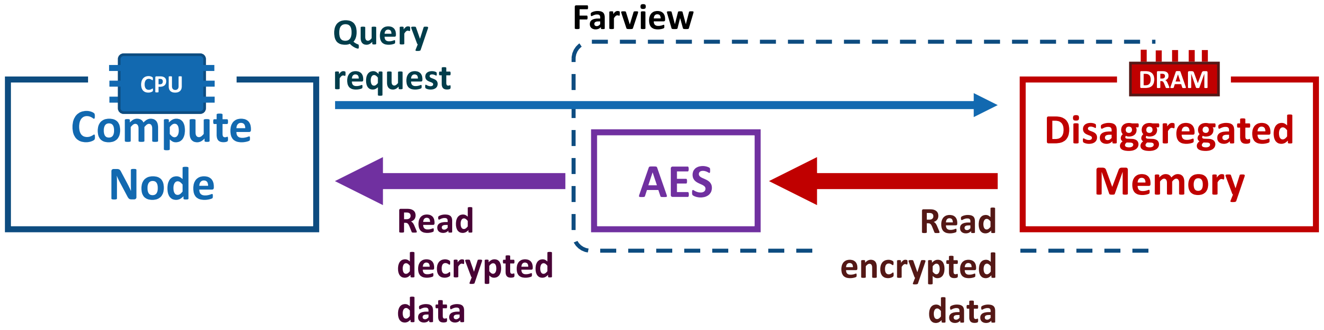
Farview

Example

- processing in Farview, simple RDMA READ operation:

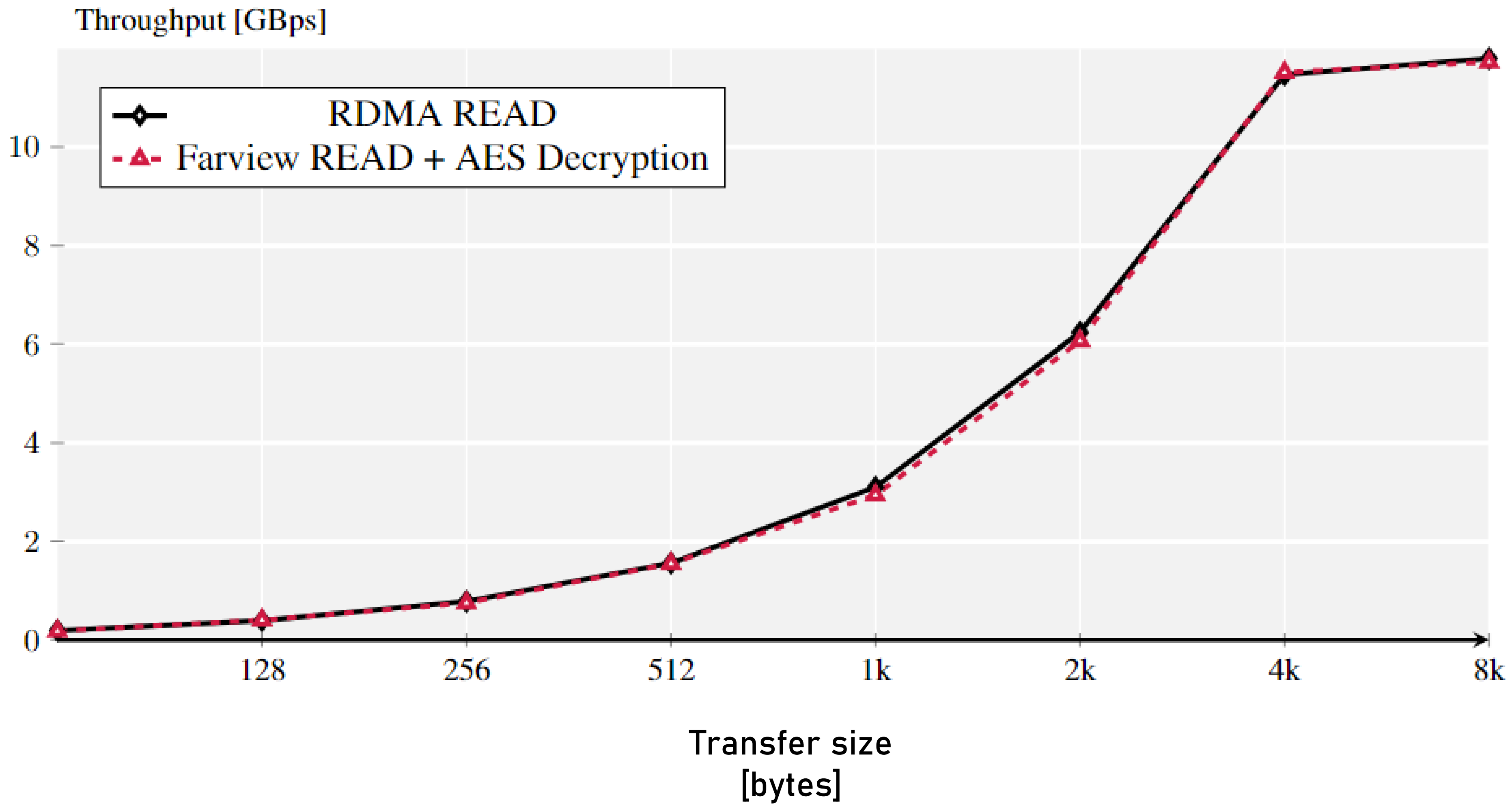


- AES decryption on the same data as it is being read:



Farview

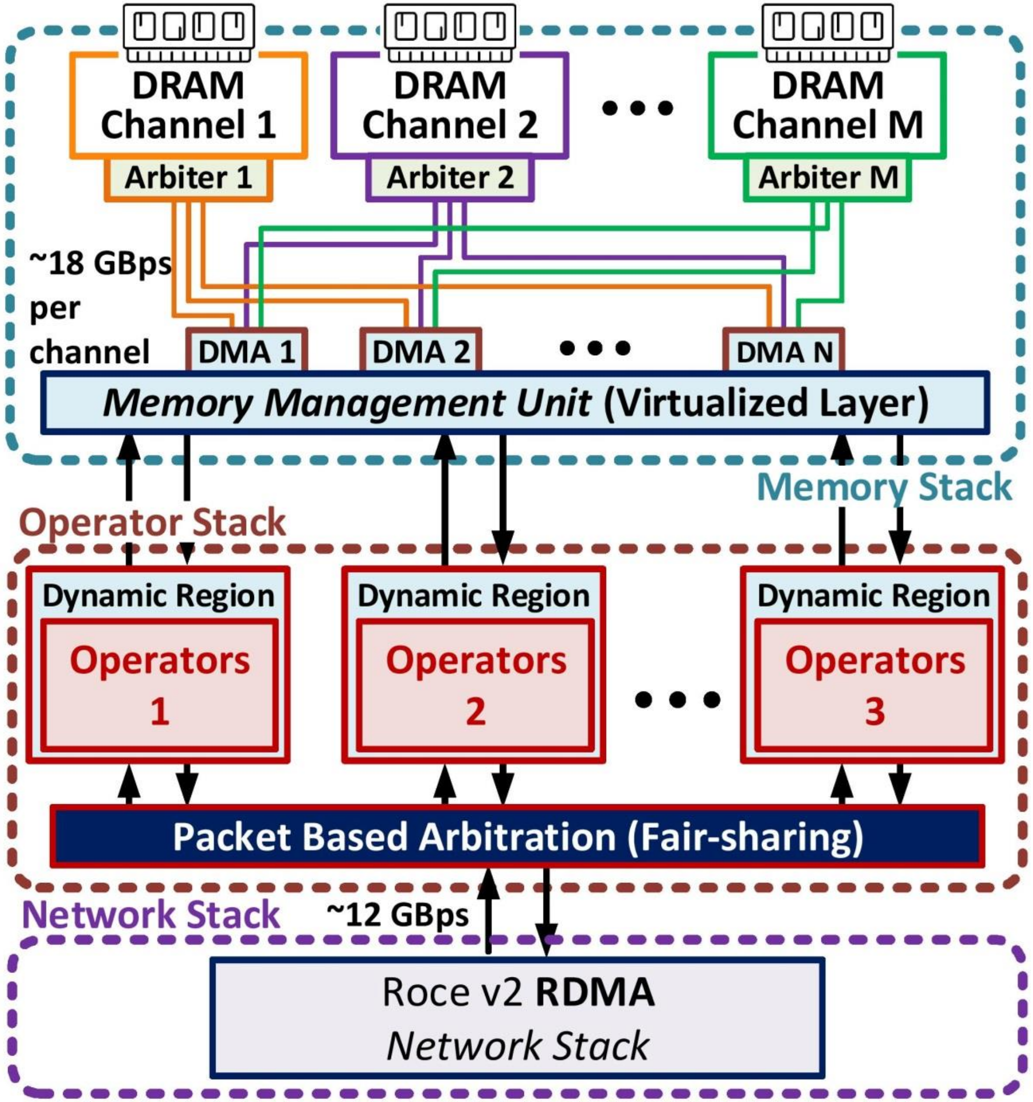
Example



Farview

Architecture

- Several components needed:
  - DRAM (HBM)
  - Memory controllers
  - Memory management unit
  - Network stack
  - Mechaniscm for concurrent access
  - Stream processing capacity
  - Mechanism to swap operators
- Three distinct layers
  - Operator stack
  - Memory stack
  - Network stack

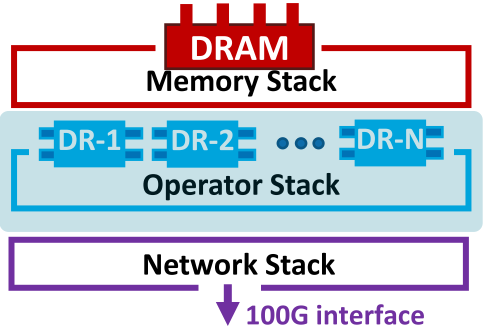


Farview

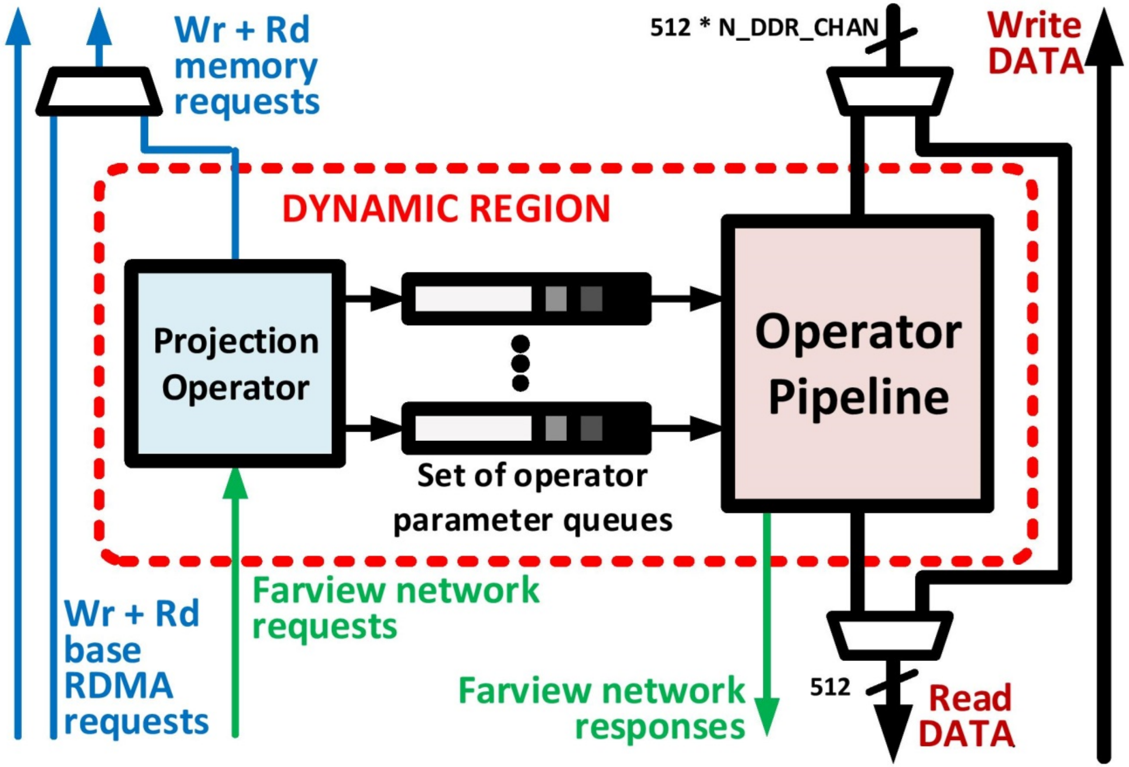
Operator stack

- Operator stack split into multiple isolated dynamic regions that operate concurrently

- Operator pipeline can execute a set of queries

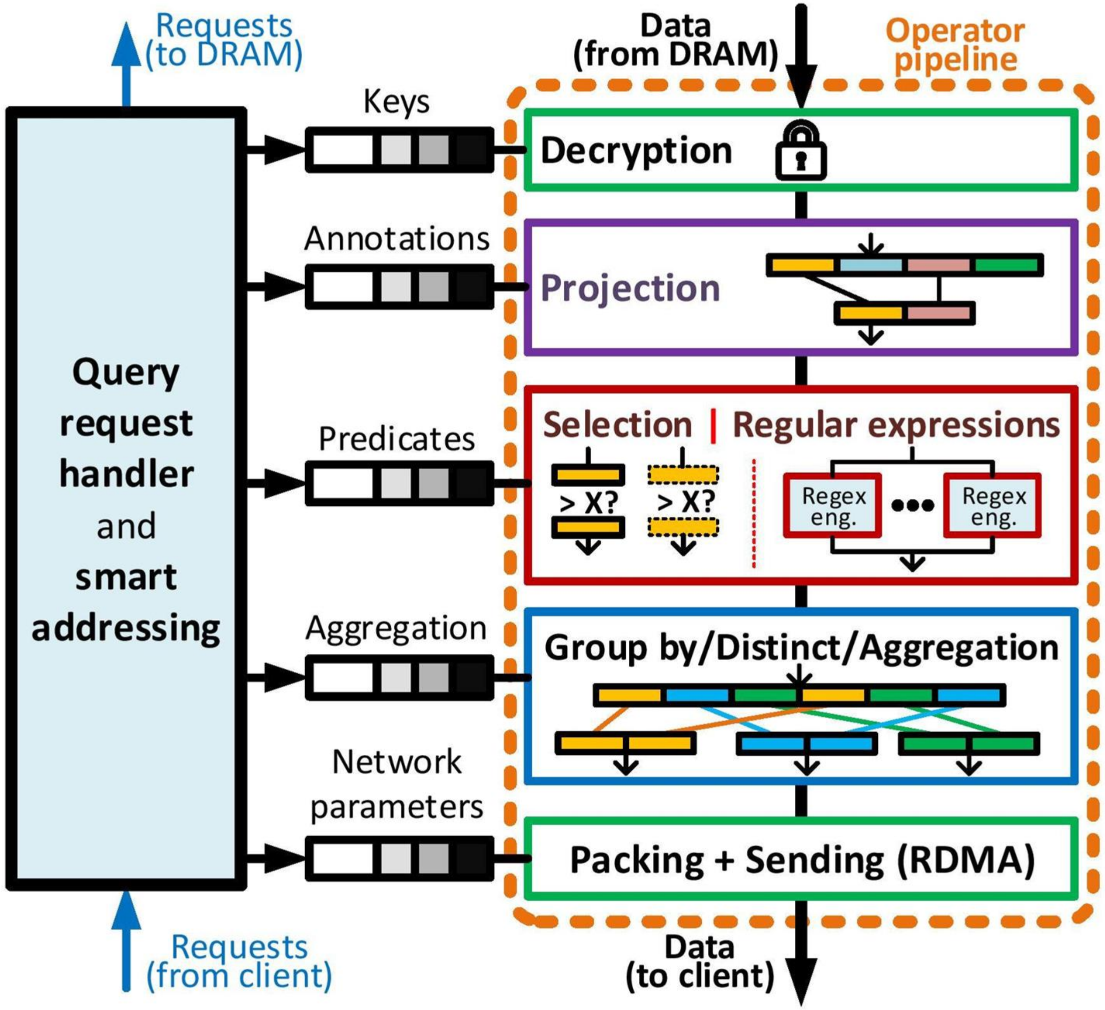


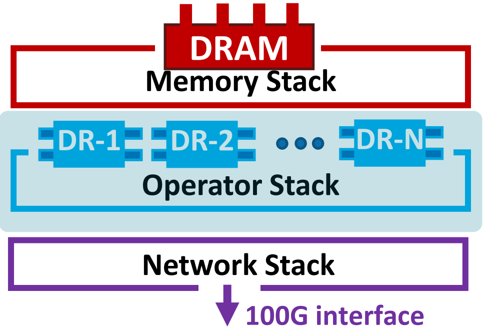
A single dynamic region and interfaces:



Farview

Operators



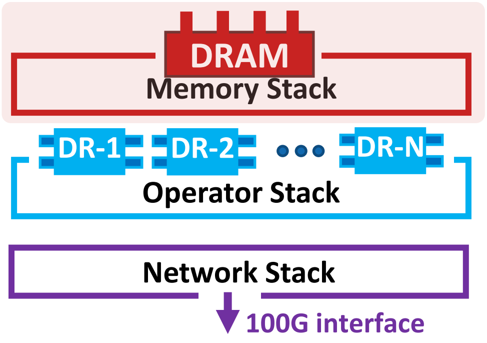


- ❖ Operator pipeline
- ❖ Farview currently supports a range of operators (row store):
  - ➢ Projection operators (smart addressing, projection)
  - ➢ Selection operators (selection, regex, vectorized selection)
  - ➢ Grouping operators (distinct, group by)
  - ➢ System operators (encryption/decryption, parsing, packing)

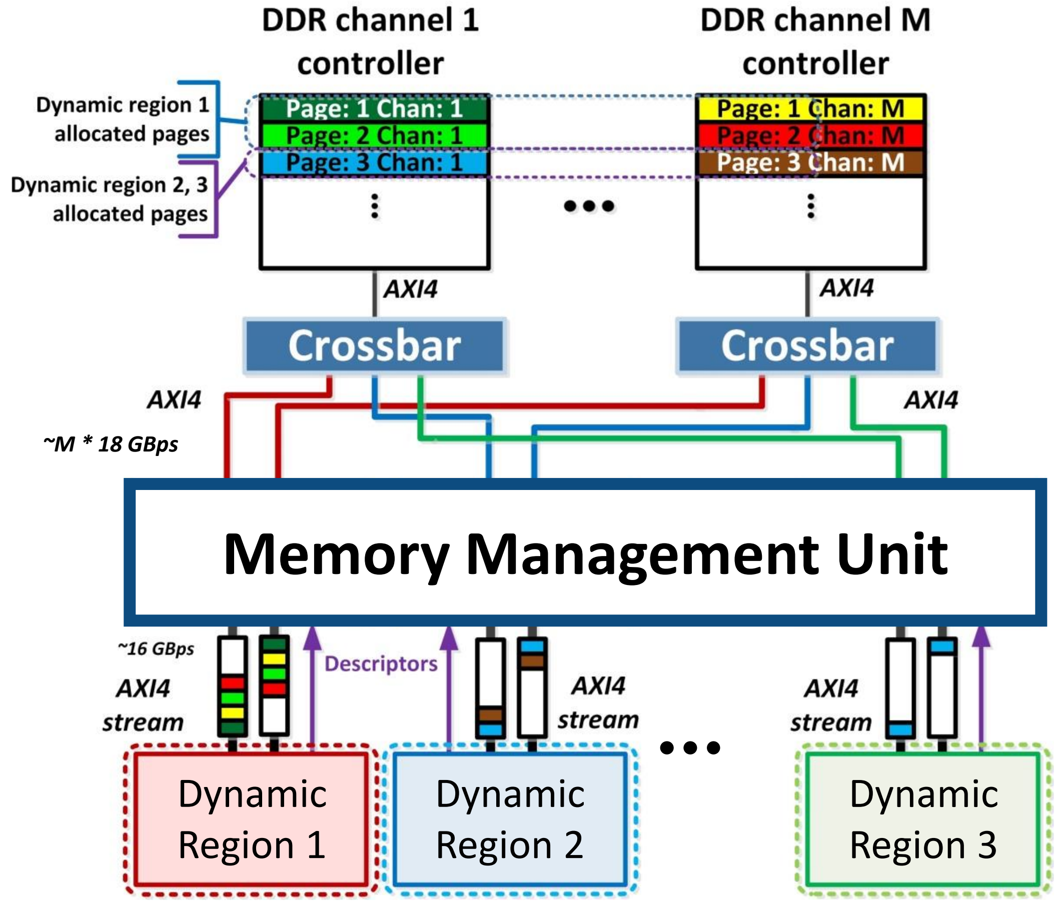
Farview

Memory stack

- Implements the actual memory buffer pool
- Organized into multiple channels
- Interleaving abstraction to aggregate the bandwidth
- Can process data at higher rates than the available network bandwidth

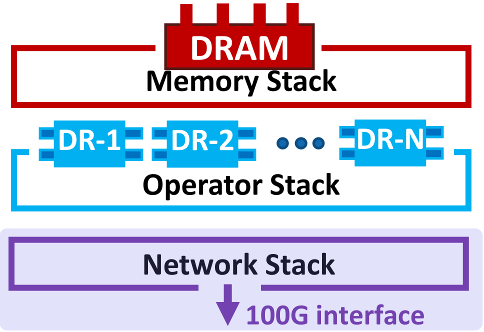


Memory stack architecture:



Farview

Network stack

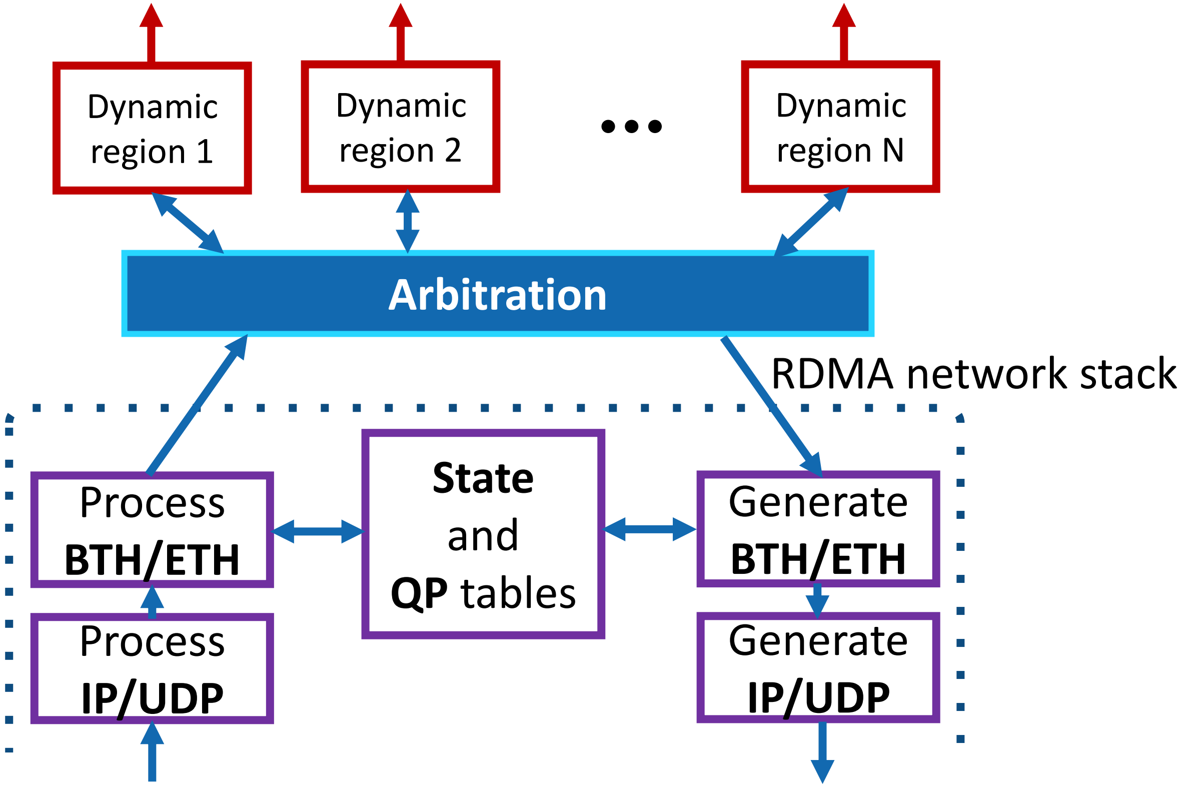


- Manages all external connections and requests for all concurrent accesses
  - Supports RoCE v2 at 100Gbps
  - Open source network stack[2]

- Special Farview verb based on InfiniBand SEND for query requests

- Comparable latencies to one-sided RDMA verbs

Network stack architecture:



[2] ***StRoM**: Smart Remote Memory, EuroSys ‘20*
David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, Gustavo Alonso

Farview

Programmatic Interface

- High level data API covering both the
  ●critical path operations and
  ●connection management operations
- Written in C++
- Intended to be used by Farview query compiler.

- ● bool openConnection(Qpair *qp, Fview *node);
- ● bool loadPipeline(Qpair *qp, int32_t opid);
- ● void tableRead(Qpair *qp, Ftable *ft);
- ● void tableWrite(Qpair *qp, Ftable *ft);
- …
- ● void farView(Qpair *qp, Ftable *ft, uint64_t *params);
- ● void select(Qpair *qp, Ftable *ft, uint64_t *proj_flags, uint64_t *sel_flags, float predicate) {

```
        …
          farView(qp, ft, params);
}
```

Farview

Implementation

- Farview supports a range of FPGA data center cards (Alveo u50, u55c, u200, *u250*, u280, Enzian)
- Low resource usage:

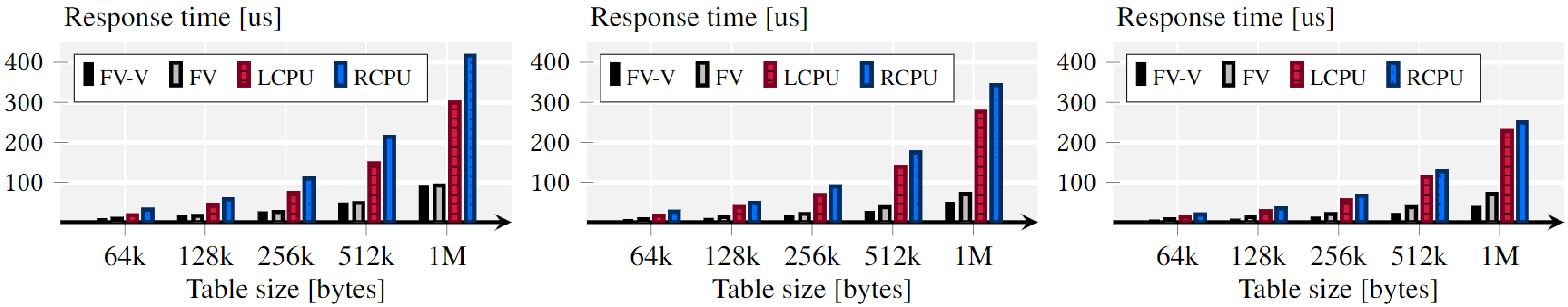
Configuration	CLB LUTs	Regs	BRAM tiles	DSPs
6 regions	24%	23%	29%	0%
Operators (per dynamic region)	**CLB LUTs**	**Regs**	**BRAM tiles**	**DSPs**
Projection / Selection	< 1%	< 1 %	0%	0%
Regex engine	2.3 %	< 1%	0%	0%
Distinct / Group by	2.1%	1.3%	8%	0%
En(de)cryption	3.6%	< 1%	0%	0%
Packing / Sending	< 1%	< 1%	0%	0%

Farview

Benchmarks

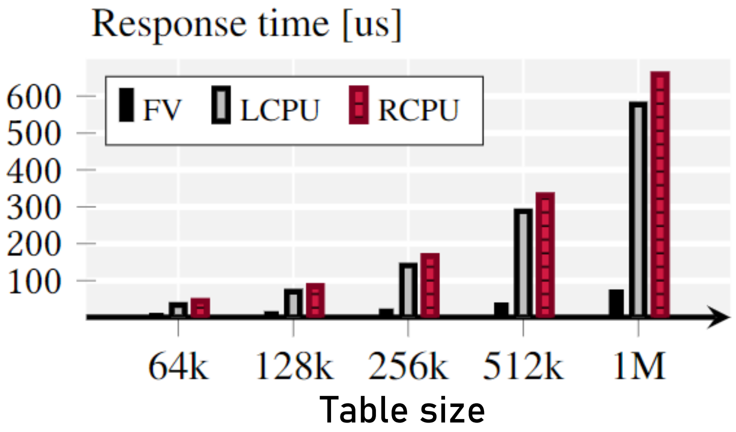
- Selection performed across different selectivity levels
- Farview outperforms two baselines (traditional database, remote memory) across different selectivity levels

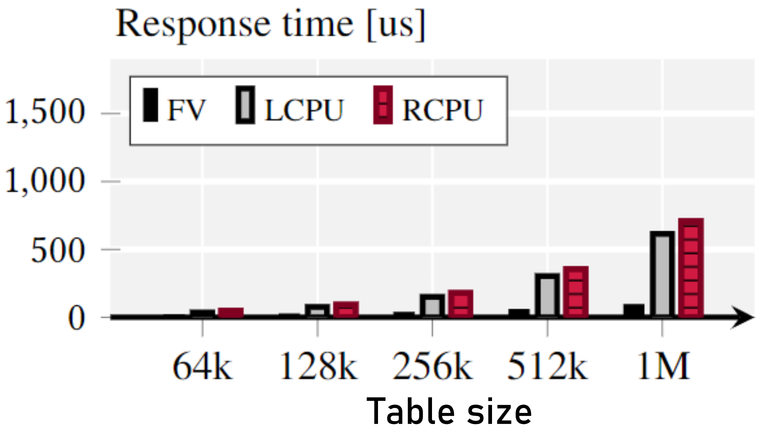
Response times for selection queries with 100%, 50% and 25 % selectivity, respectively:

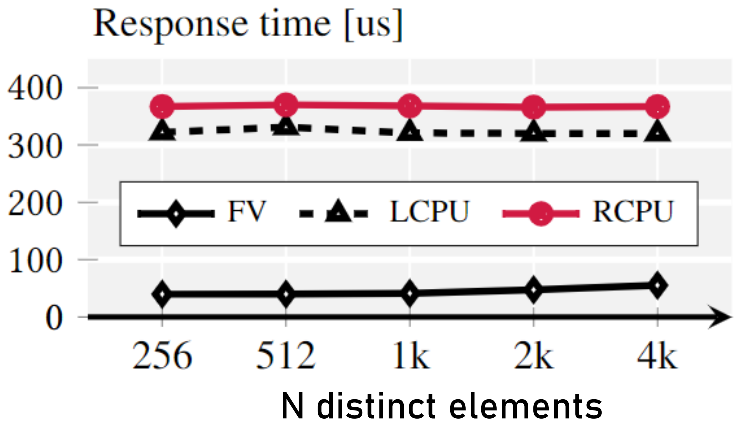


Farview

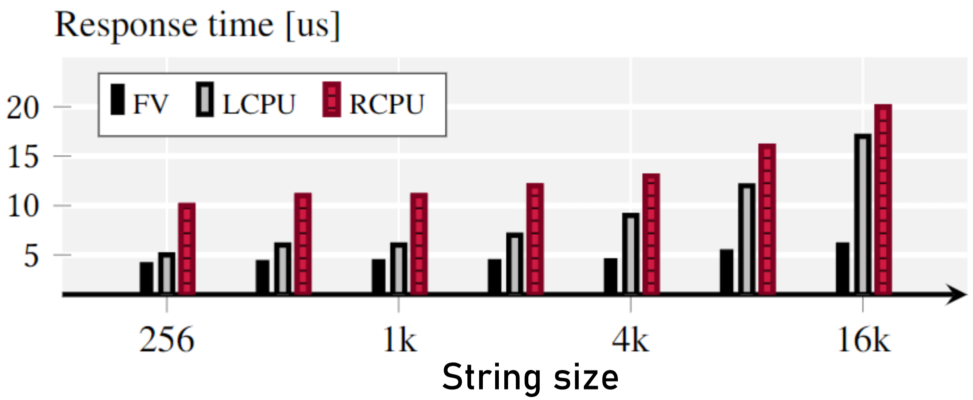
- Distinct, Group by, Group by (sweep by n. of distinct el.)

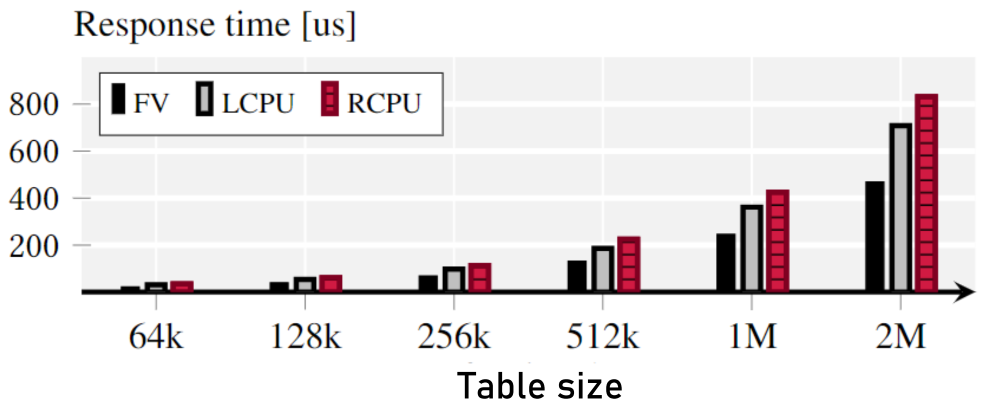






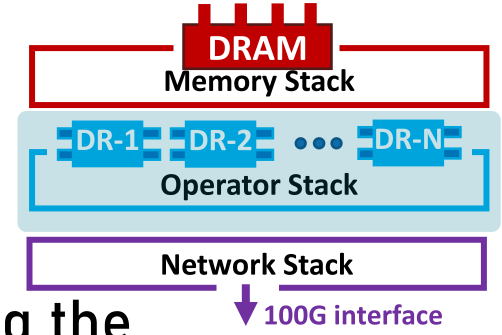
- Regex and concurrent queries (3x)



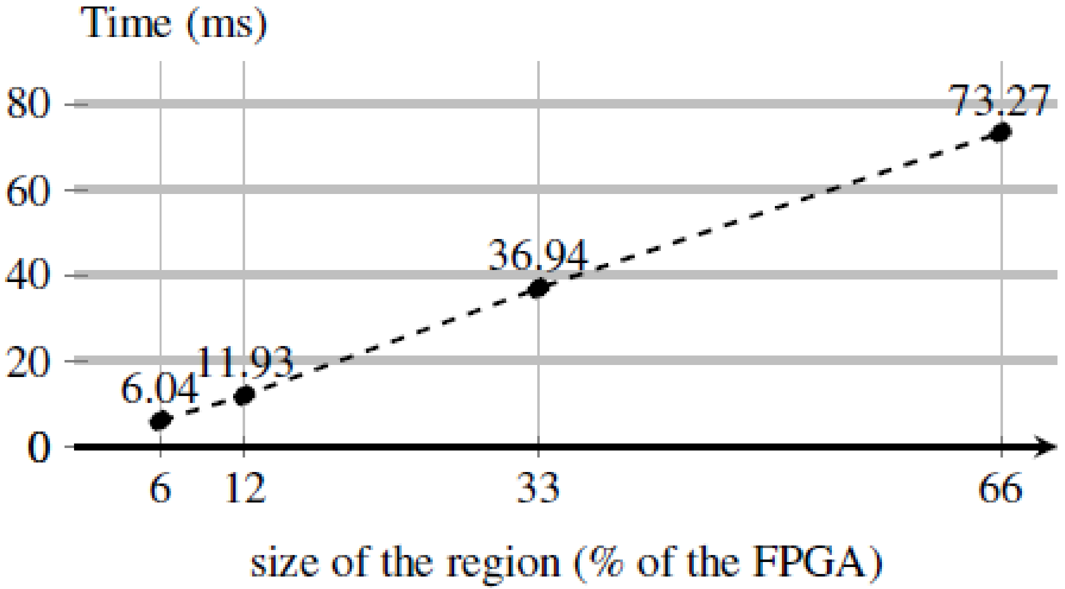


Farview

Operator Pipeline Swap



- Operator pipelines can be swapped during *runtime* without affecting the integrity of the system
- Gives Farview a much needed flexibility in comparison to traditional accelerators
- Swap time in the order of milliseconds:



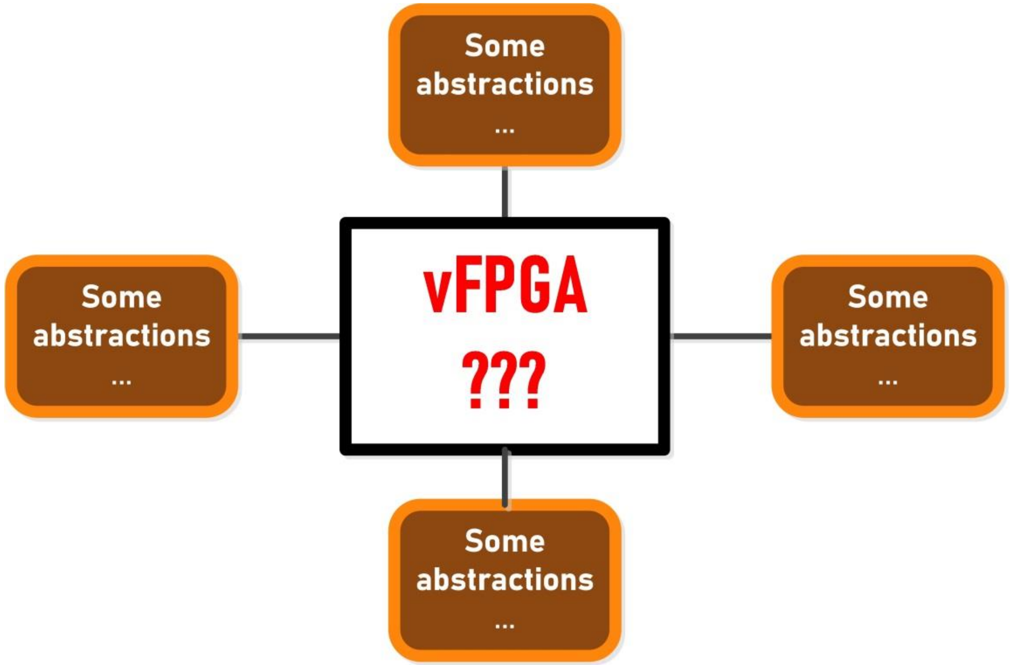
Farview

Current work

- Additional operators …
- Interaction with the storage layer (deduplication …)
- Scale out …
- Serverless (end-to-end latencies in s range)
- Proper database frontend (Modularis)

What about the vFPGA?

- HDL … (SVerilog, VHDL, HLS, OpenCL ,…)



Some abstractions …

Some abstractions …

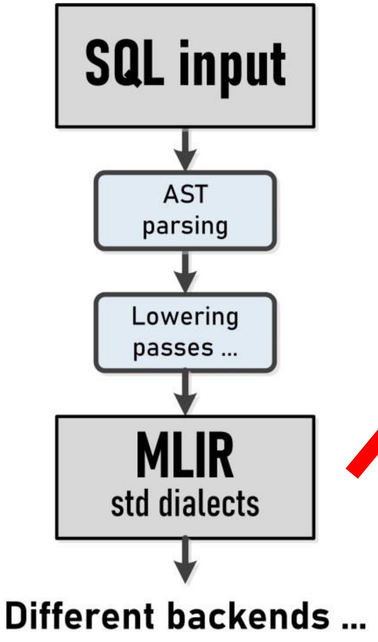
vFPGA ???

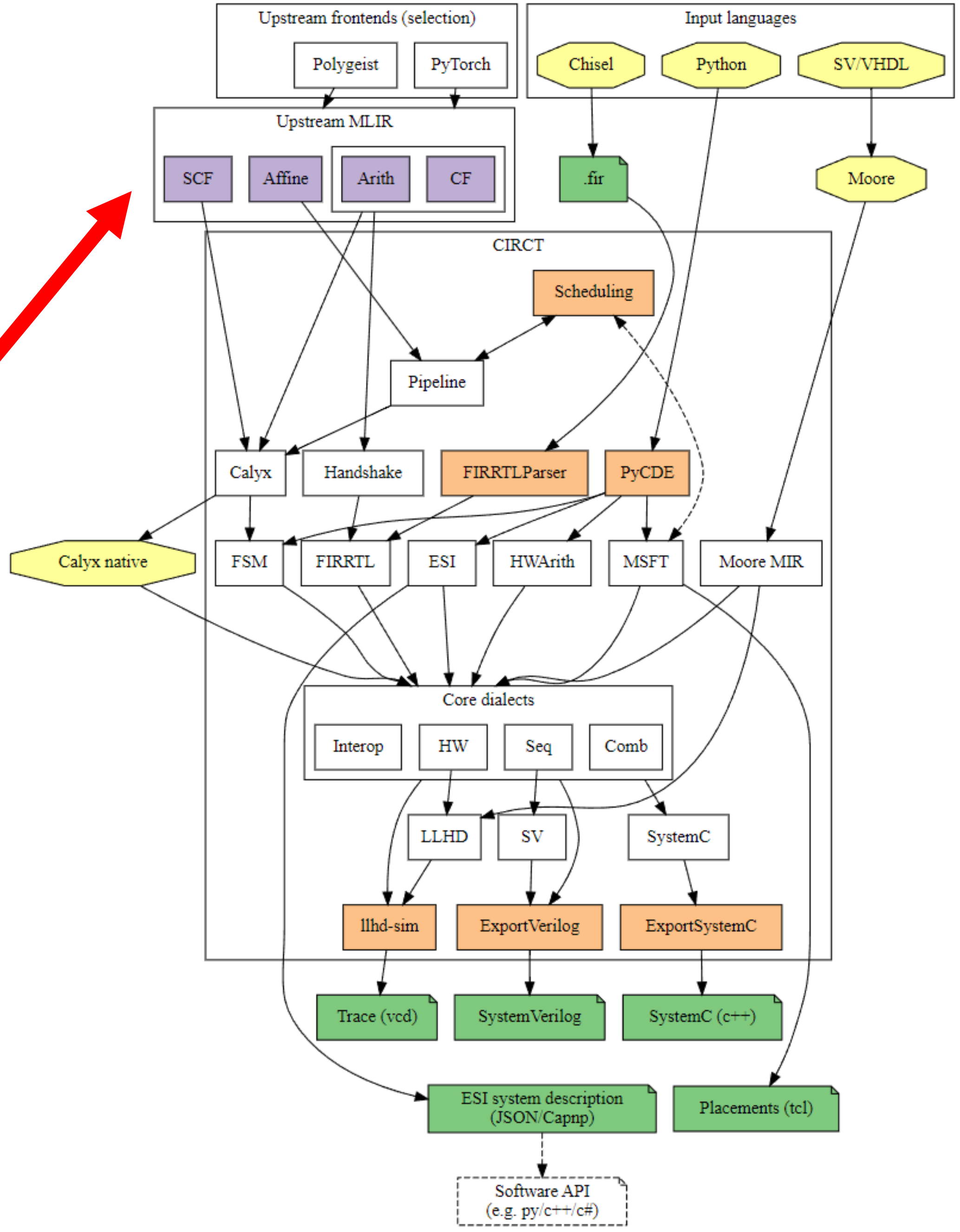
Some abstractions …

Some abstractions …

MLIR compiler

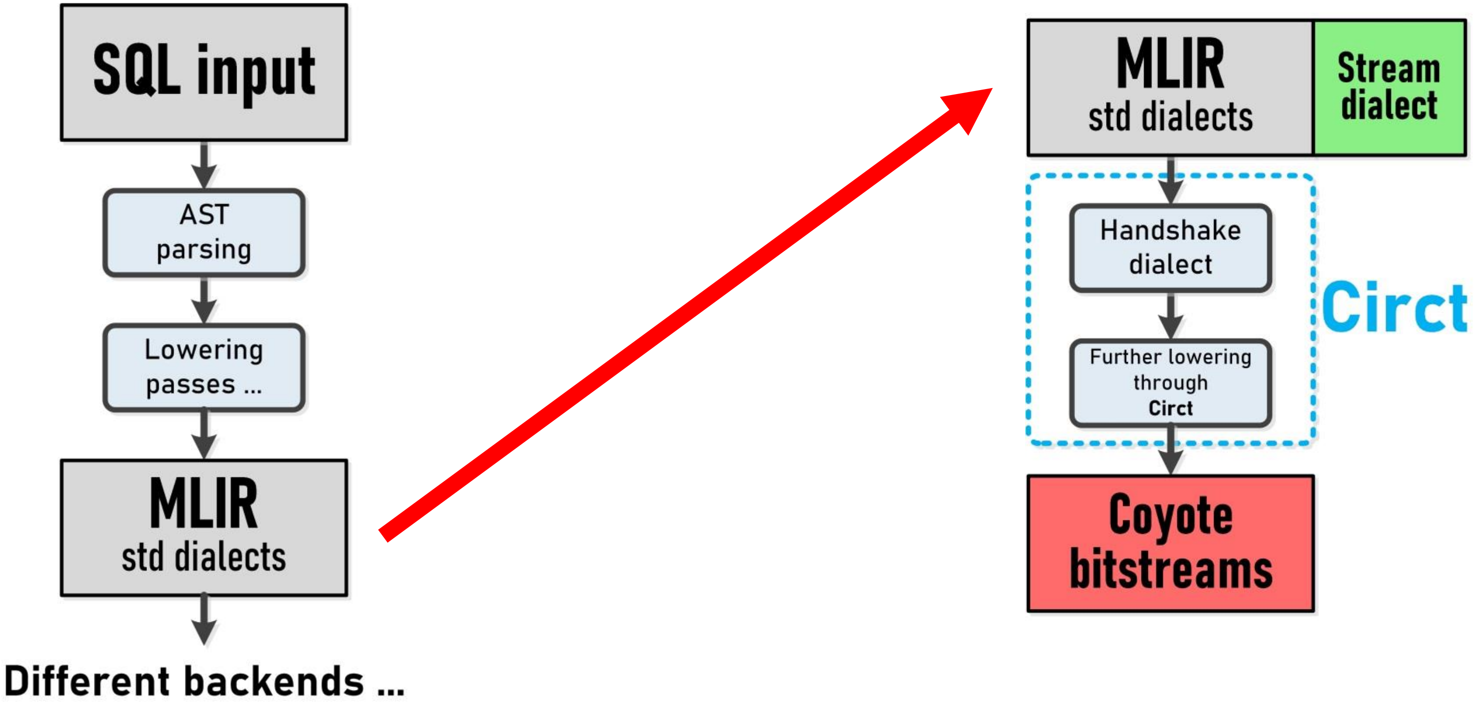
- MLIR is a novel approach to building a compiler
- CIRCT built in MLIR, targets HDL (*https://github.com/llvm/circt.git*)
- Modularis infrastructure being ported to MLIR





Modularis + Farview

- 2 Master thesis projects
- Stream-dialect => Coyote-CIRCT (*https://github.com/fpgasystems/Coyote-CIRCT.git*)



Questions?