arm

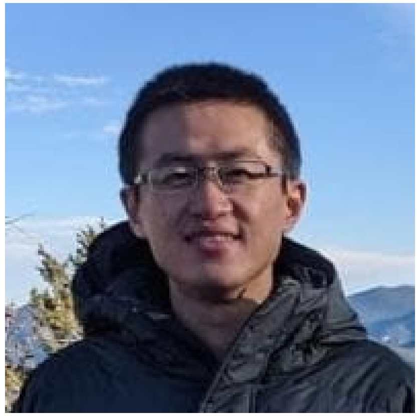
Bring the BitCODE: Moving Compute and Data in Distributed Heterogeneous Systems

UCF Annual Meeting 2022

Luis E. Peña

21 September 2022

Who?



Wenbin Lü
Stony Brook

Luis E. Peña
Arm Research

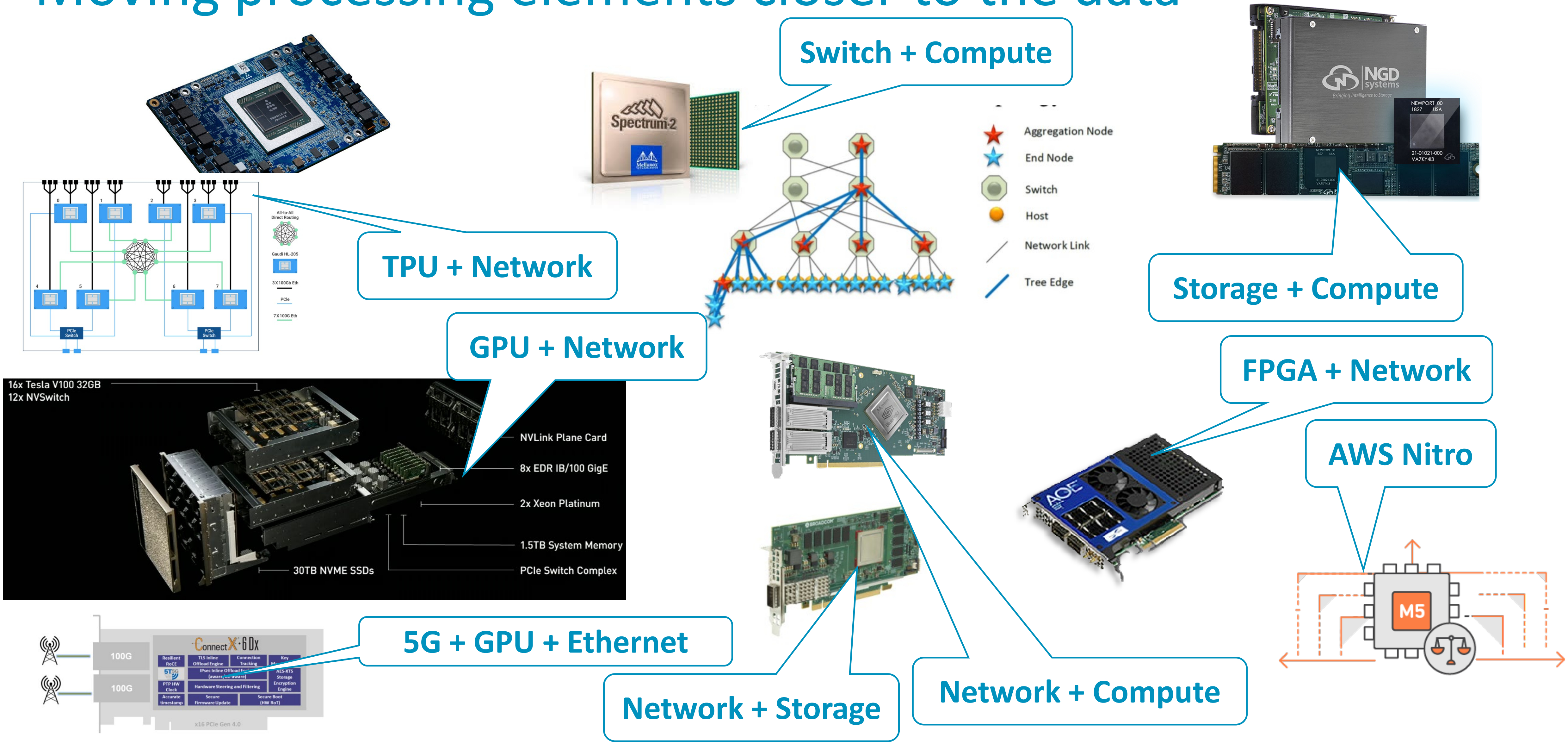
Pavel Shamis
Nvidia

Valentin Churavy
MIT

Steve Poole
LANL

Barbara Chapman
Stony Brook

Moving processing elements closer to the data



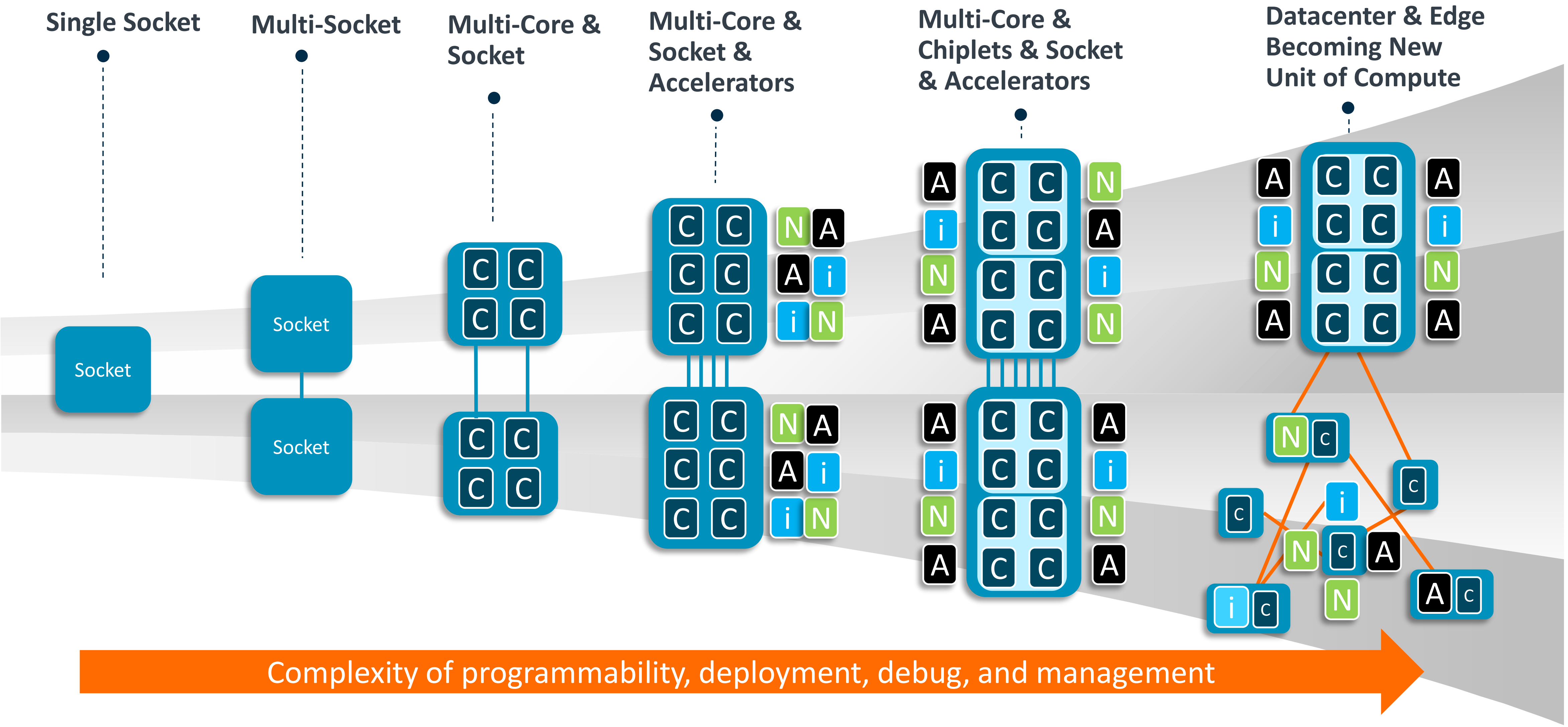
Motivation

- Target heterogeneous devices on the network
  - DPU (Data Processing Units)
  - CSD (Computational Storage Devices)

- Devices have typically: Limited storage, connected through RDMA capable interconnect

- How do we program these devices:
  - Preload binary
  - Over the network

Complexity of Modern Distributed Systems



The *Two-Chains* Framework

Framework underpinning the *ifunc* API

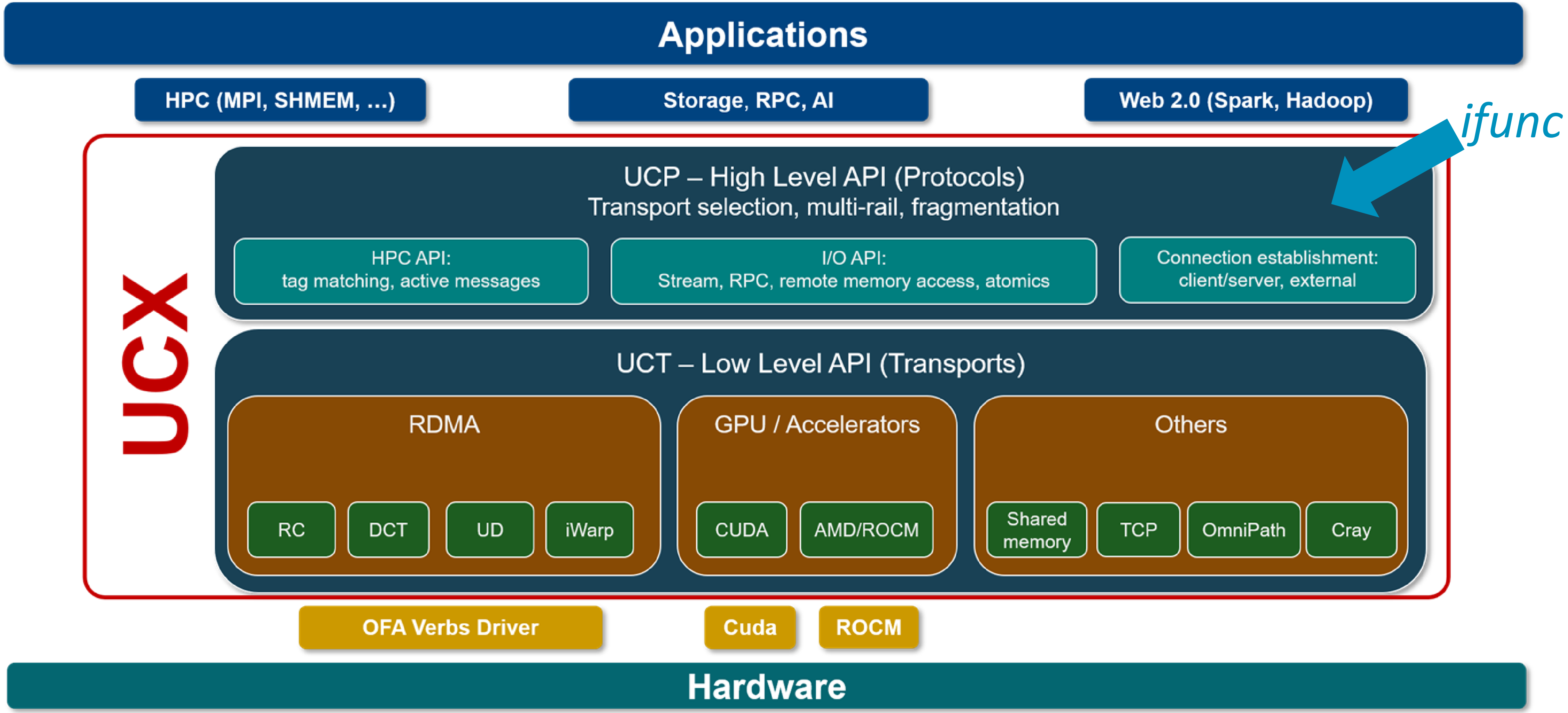
- Provides packaging, transfer and execution of functions on local and remote processes
  - Functions are loaded as dynamic libraries
  - Messages contain binary code and data
- Fast, lightweight and portable
  - Low latency & high throughput
  - Functions are written in regular C code
  - Works on CPUs, DPUs and CSDs
- Extension of the UCX framework
  - *Two-Chains* leverages UCP put semantics

Two-Chains: High Performance Framework for Function Injection and Execution, IEEE CLUSTER 2021
Authors: Megan Grodowitz, Luis E. Peña, Curtis Dunham, Dong Zhong, Pavel Shamis & Steve Poole

UCX Programming Interface for Remote Function Injection and Invocation, OpenSHMEM 2021
Authors: Luis E. Peña, Wenbin Lü, Pavel Shamis & Steve Poole

Bring the BitCODE -- Moving Compute and Data in Distributed Heterogeneous Systems, IEEE CLUSTER 2022 (this work)
Authors: Wenbin Lü, Luis E. Peña, Pavel Shamis, Valentin Churavy, Barbara Chapman & Steve Poole

Where *ifunc* fits



Source: https://openucx.org/

`ifunc` Basics

- A C/Julia function is compiled and shipped to a remote process in the form of an *ifunc* message

- The message also contains a set of arguments (aka payload) for the *ifunc*

- The *ifunc* can access code and/or data on the target process (`target_args`)
  - The target arguments are passed to the function by the target process
  - The *ifunc* can invoke local functions on the target

```
void foo_main(void *payload, size_t payload_size, void *target_args)
```

Bring the Bitcode! (Three-Chains)

Extending the Two-Chains *ifunc* work by:

- Removing the need of the shared library to be present on the target
- Using LLVM bitcode as an intermediate format
- Caching the bitcode
- Demonstrating that the approach is extendable to a high-level dynamic language Julia

Julia: Yet another high-level language?

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end

julia> mandel(complex(.3, -.6))
14
```

Julia: Yet another high-level language?

Typical features

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

Unusual features

Great performance!

JIT AOT-style compilation

Most of Julia is written in Julia

Reflection and metaprogramming

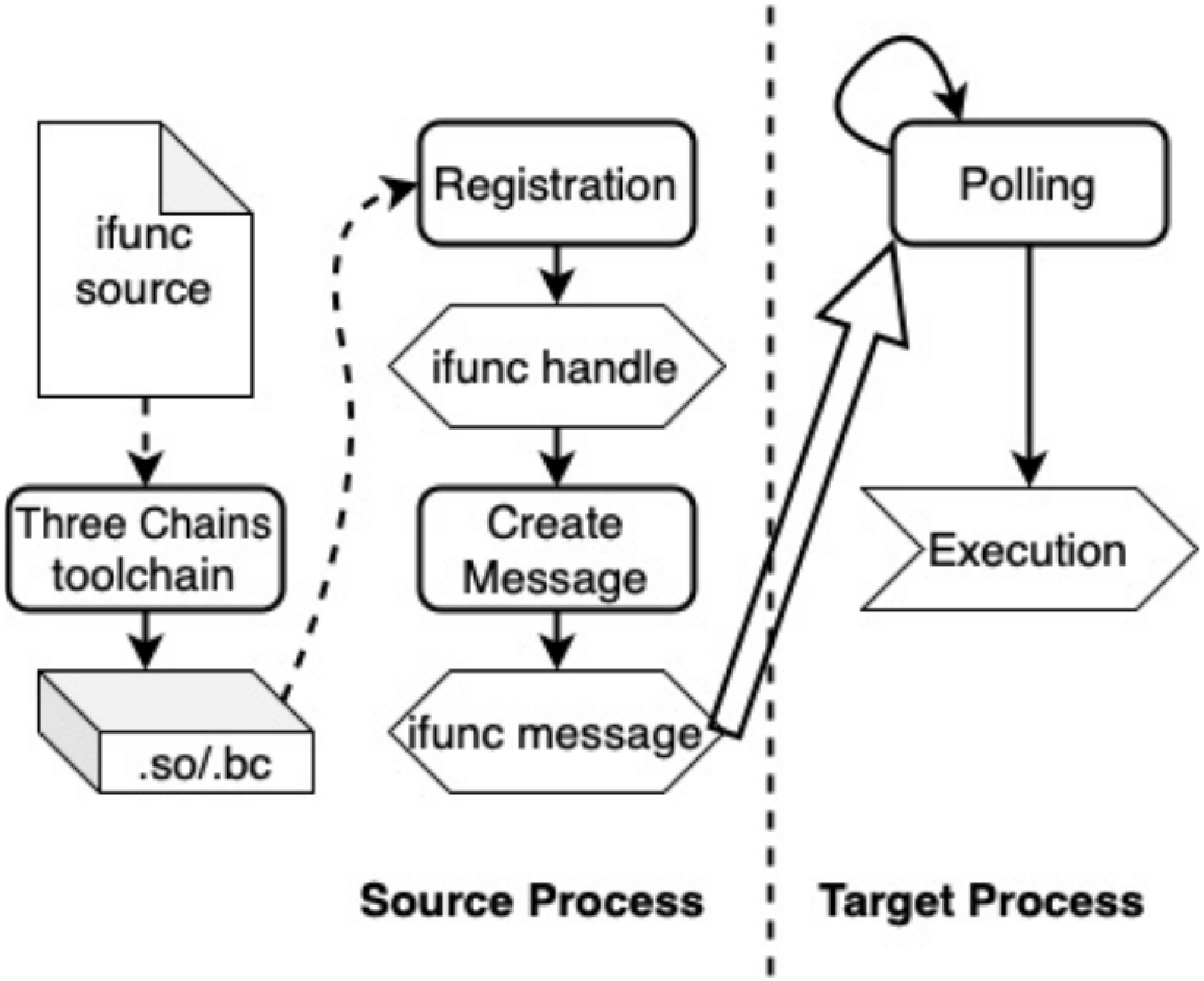
Et tu Julia?

1. JIT compiler based on LLVM
2. UCX bindings
3. Used in HPC & ML, may open up interesting applications
4. Demonstrates generality (and limitations) of our approach.

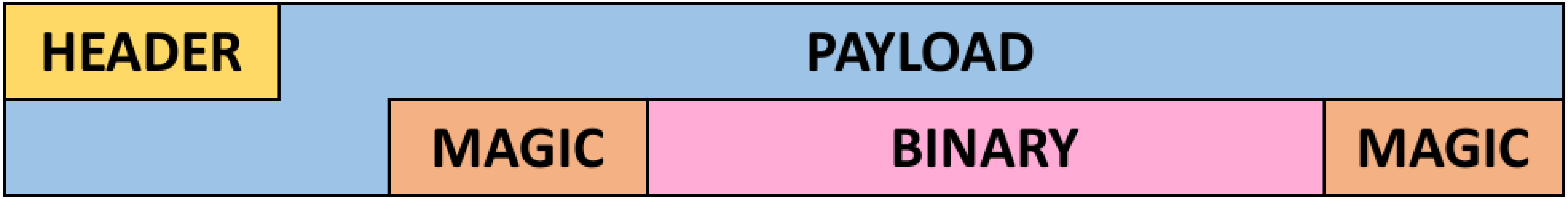
Sample Julia ifunc

```julia
function init(source_args::Ptr{Cvoid}, source_args_size::Csize_t,
              payload::Ptr{Cvoid}, payload_size::Csize_t)::Cint
    result_src = Base.unsafe_convert(Ptr{UInt64}, source_args)
    result_pay = Base.unsafe_convert(Ptr{UInt64}, payload)

    Base.unsafe_store!(result_pay, Base.unsafe_load(result_src))

    return Cint(0)
end

function main(payload::Ptr{Cvoid}, payload_size::Csize_t,
              target_args::Ptr{Cvoid})::Cvoid
    result_pay = Base.unsafe_convert(Ptr{UInt64}, payload)
    result_tgt = Base.unsafe_convert(Ptr{UInt64}, target_args)

    Base.unsafe_store!(result_tgt, Base.unsafe_load(result_pay))

    return nothing
end
```

Three-Chains workflow



Binary based *ifunc*



1. Compile program to shared library
2. Load shared object and pack it into the binary section
3. Perform run-time symbol resolution on remote system / remote dynamic linking

Issues:

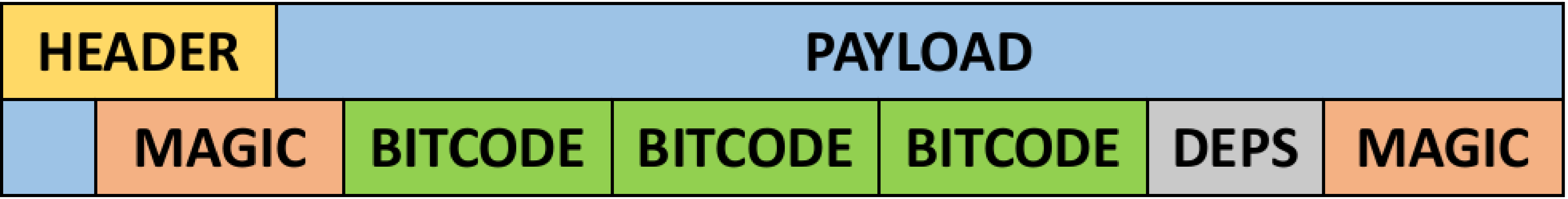
- Architecture dependent
- Remote-dynamic linking is complicated and must be implemented for each target

Could we not just send source-code?

Instead of sending over shared-object file we could send the source-code

1. Julia's `Distributed.jl` actually does so.
2. Complicated for C
   a. Need a compiler present
   b. Headers/source code are not trivial to locate & large
   c. Much higher initial latency

Heterogenous bitcode



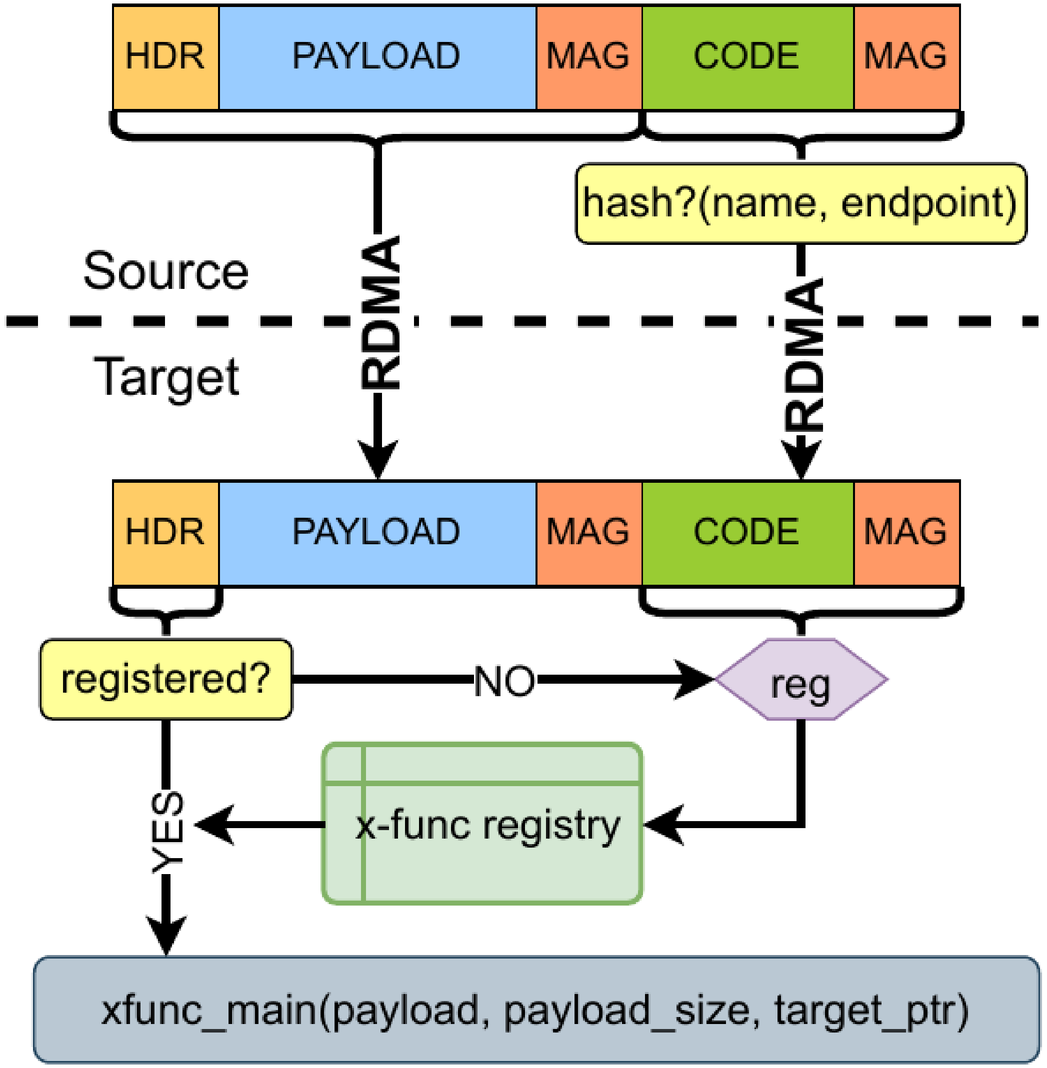
1. Use LLVM bitcode as a serialization format
2. Allows easily for multiple-architectures to be present
3. LLVM ORC JIT compiles bitcode to machine code and performs linking
   a. Also performs symbol resolution for us
   b. DEPS: Contains names of libraries we should load beforehand

Self propagation / caching

- Compile *ifunc* once
  - Send across the network
  - “Fat” bitcode — for each target architecture
  - Myth of the target-independent LLVM bitcode
    - Clang generates-target specific IR
    - LLVM optimization use target information to choose vector width etc

- Latency trade-offs
  - Send binary
  - **Send late-opt bitcode**
  - Send pre-opt bitcode
  - Send source code

Caching


HDR
PAYLOAD
MAG
CODE
MAG
hash?(name, endpoint)
Source
RDMA
RDMA
Target
HDR
PAYLOAD
MAG
CODE
MAG
registered?
NO
reg
YES
x-func registry
xfunc_main(payload, payload_size, target_ptr)

julia Integration

1. Julia is loaded on all targets

2. Reusing Julia GPUCompiler to collect a LLVM module containing the IFunc

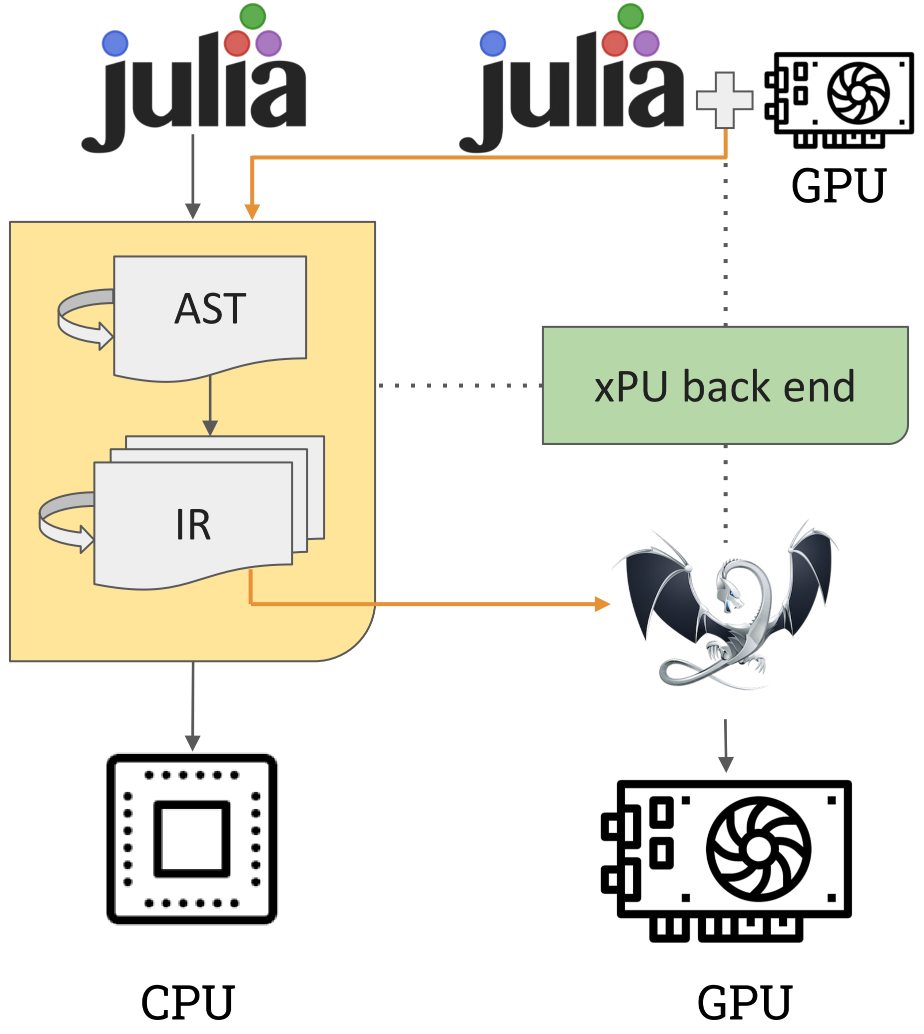
3. Using UCX.jl to setup program and IFunc/s

Caveats:

1. Julia currently doesn't have support for cross-compilation.

2. Set of Julia constructs in IFuncs are limited
   a. No dynamic-dispatch
   b. Runtime interactions are supported

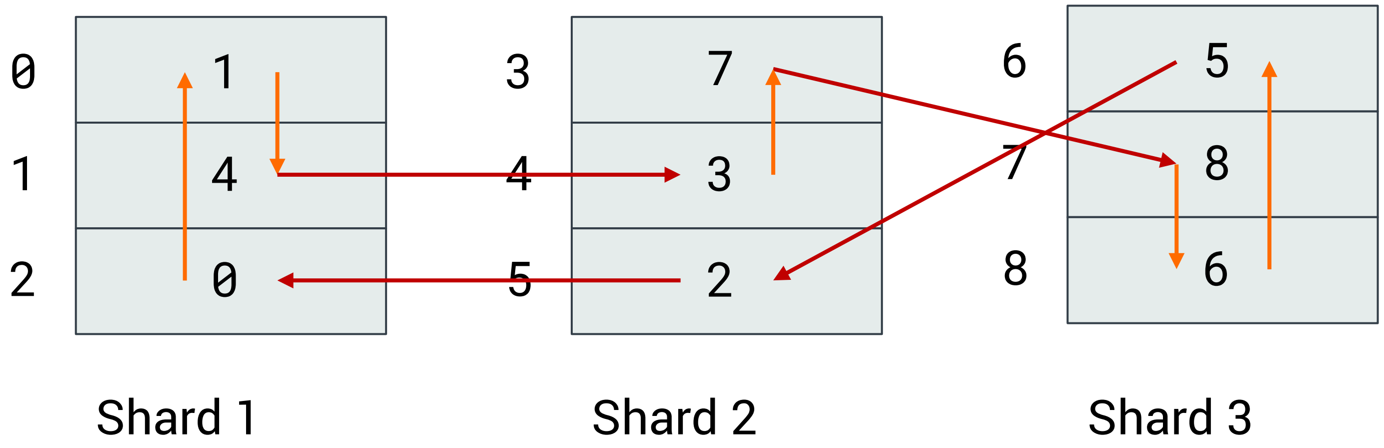
3. Julia can be too aggressive and embed pointers to global data into generated IR.



Effective Extensible Programming: Unleashing Julia on GPUs (doi:10.1109/TPDS.2018.2872064)

arm

Benchmark — Pointer chase



Parameters:

- Number of shards
- Depth (length of chase)

1. Random (but consistent across runs) initialization
2. Local work (orange), remote work (red)
3. Number of network jumps is important

Benchmark — Pointer chase

Three different conditions:

1. Pseudo-AM (Active message)
   Pre-installed function on target side — as if code was already present

2. RDMA GET
   Client process loads values via RDMA GET — no local work possible

3. *ifunc* based
   Dynamically propagated and JIT compiled/linked

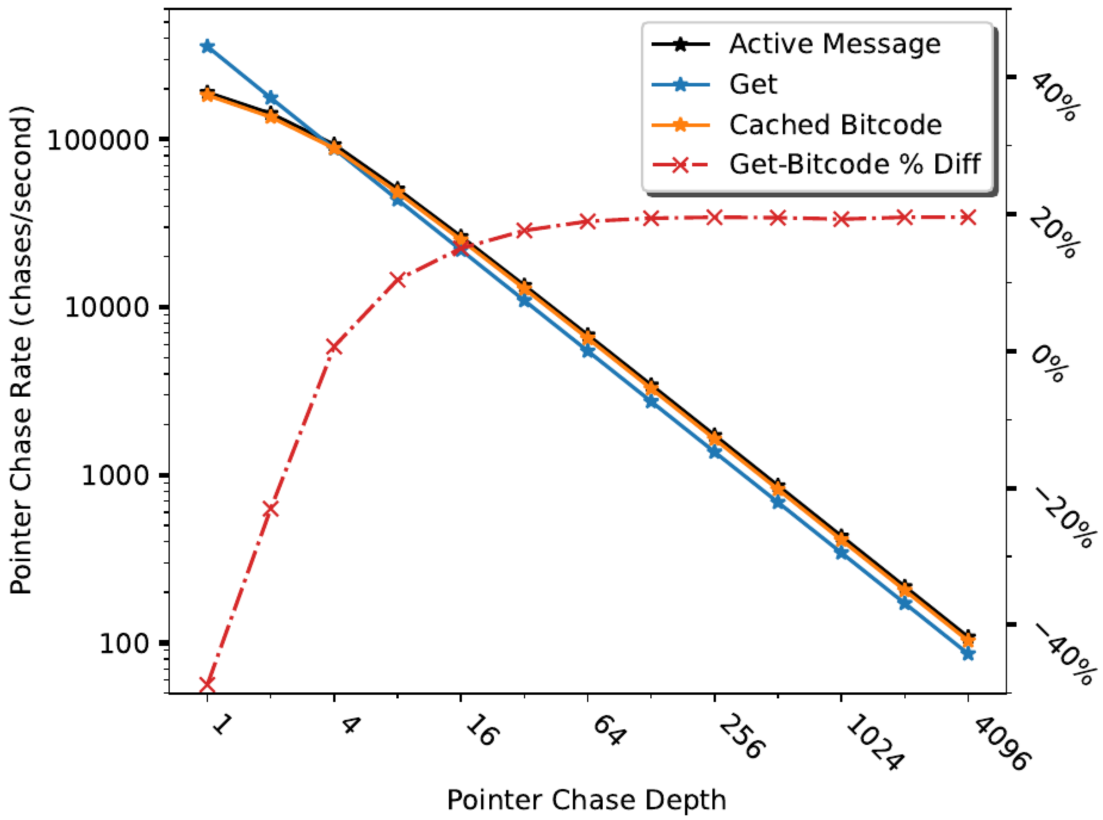
Test machines

- Thor 36-node Cluster (hosted by the HPC Advisory Council)
  - Dual Socket Intel Xeon 16-core CPUs E5-2697A with 256GB DDR4 memory
  - ConnectX-6 HDR 100Gb/s InfiniBand
  - BlueField-2 HDR 100Gb/s DPU
    - 8x Arm Cortex-A72 with 16GB DDR4 memory
  - Configurations
    - Xeon Client + BF2 Server
    - Xeon Client + Xeon Server

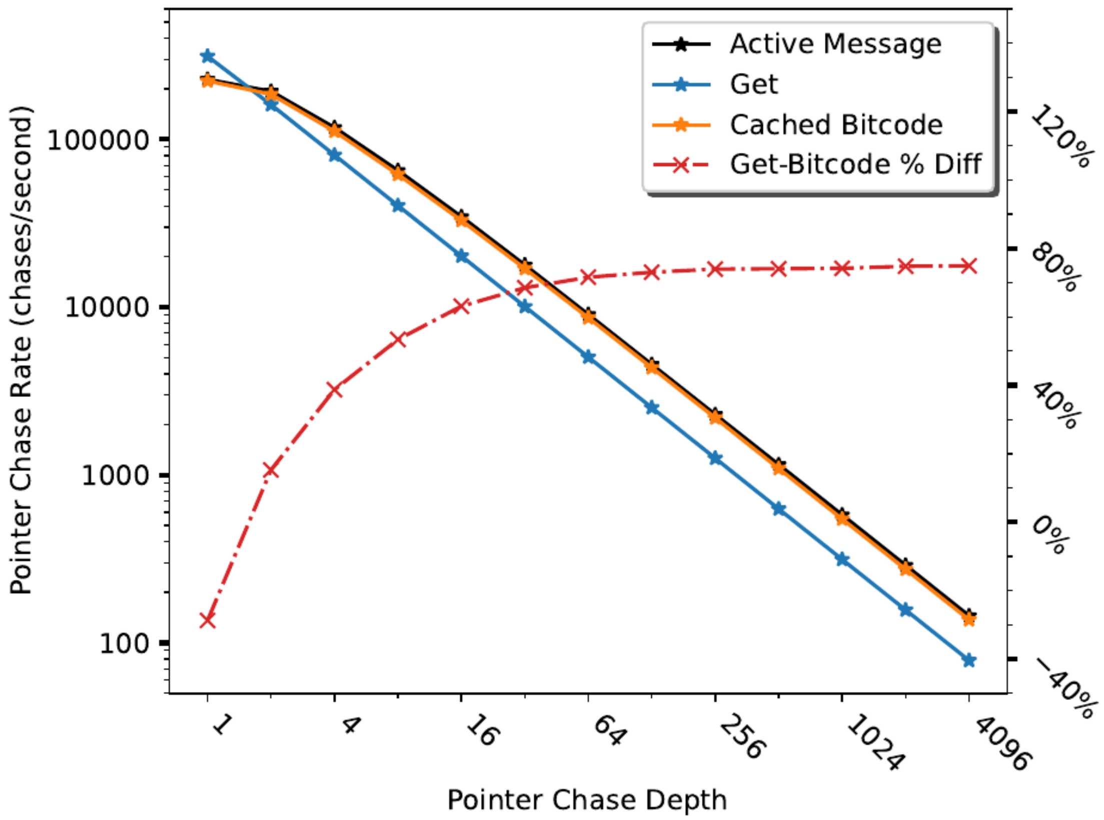
- Ookami 174-node Cluster (hosted by Stony Brook University)
  - 48-core Fujitsu A64FX FX600 with 32GB HBM memory
  - ConnectX-6 HDR 100Gb/s InfiniBand

Results: Xeon-BF vs Xeon

Varying depth



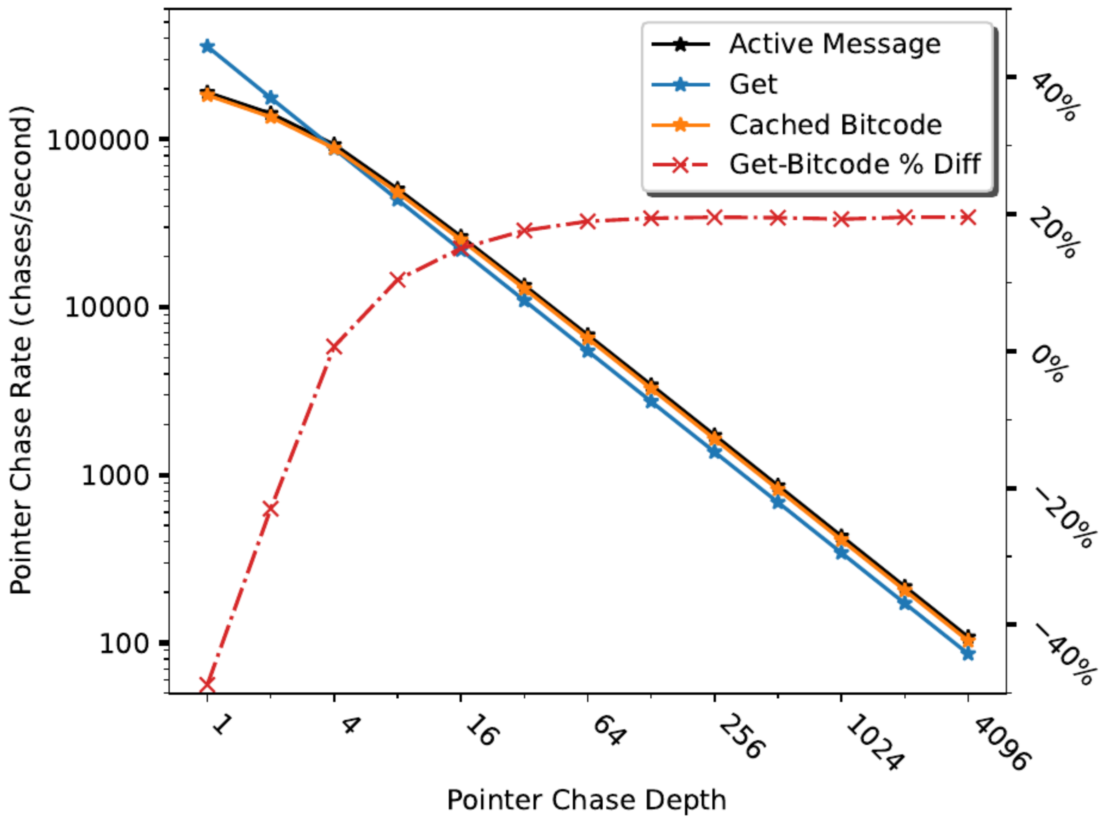
Thor 32-Server **C/C++**
(Xeon Client and BF2 Servers)



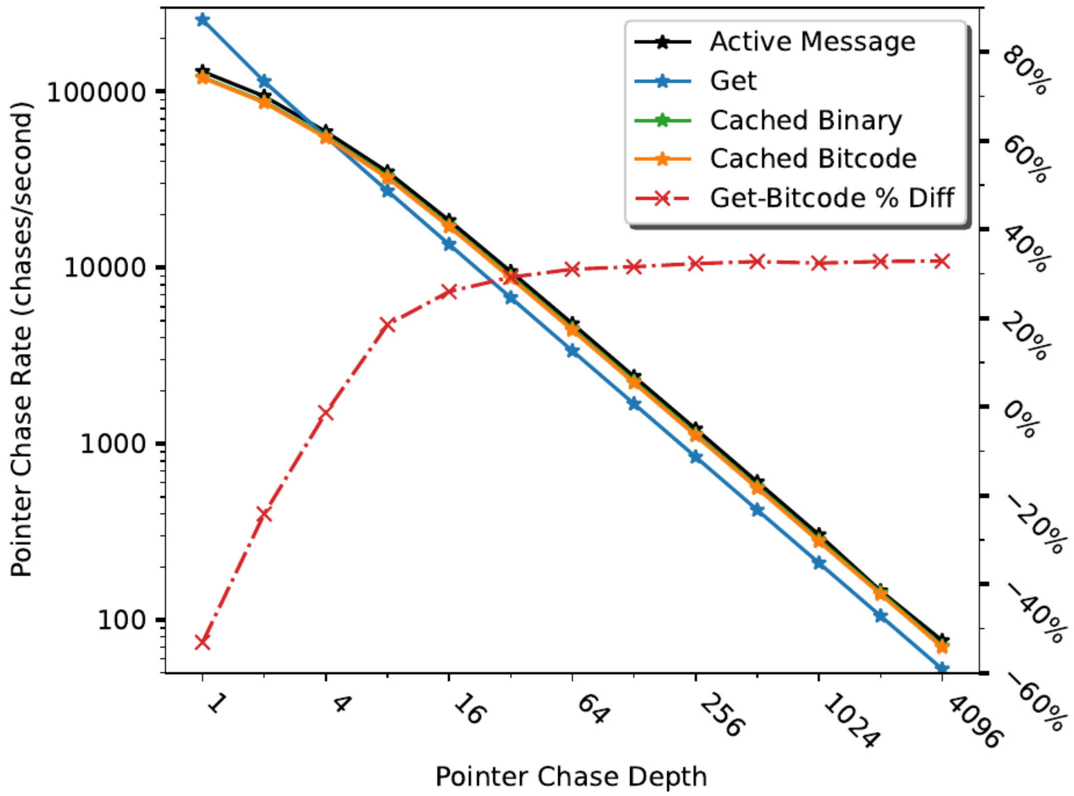
Thor 16-Server **C/C++**
(Xeon Client and Servers)

arm

Results: Xeon-BF vs A64FX

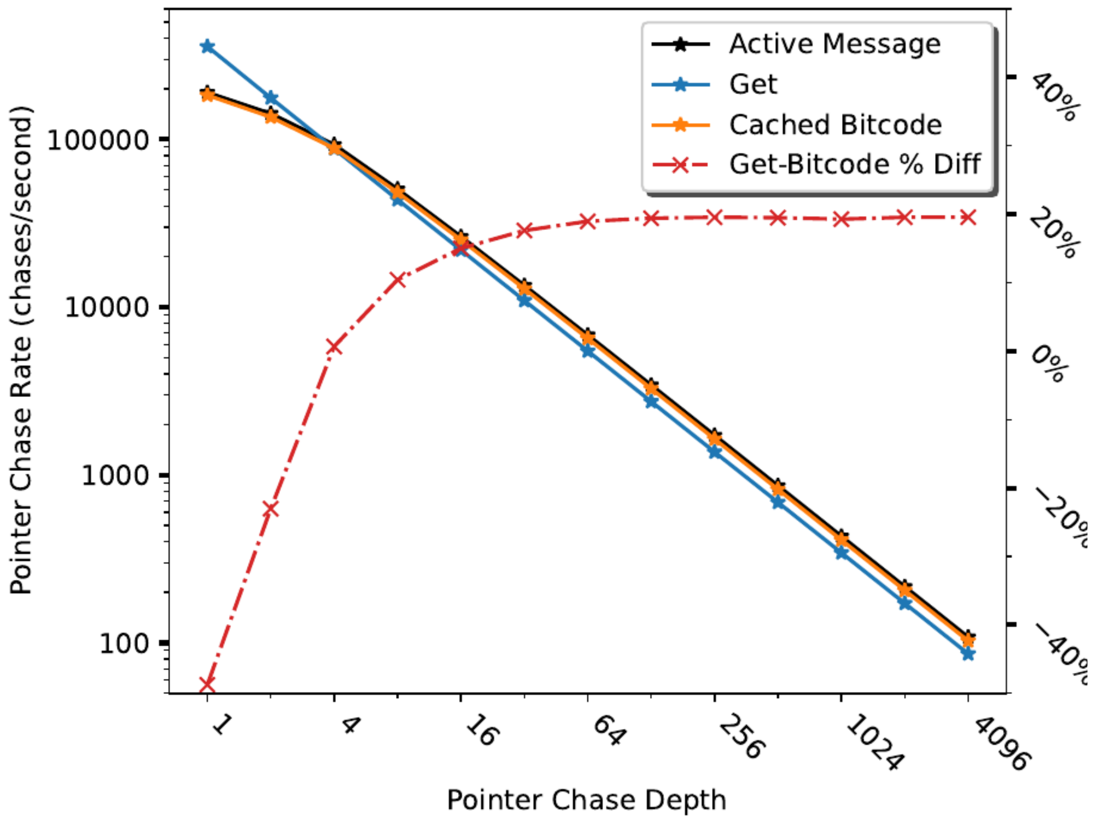


Thor 32-Server **C/C++**
(Xeon Client and BF2 Servers)

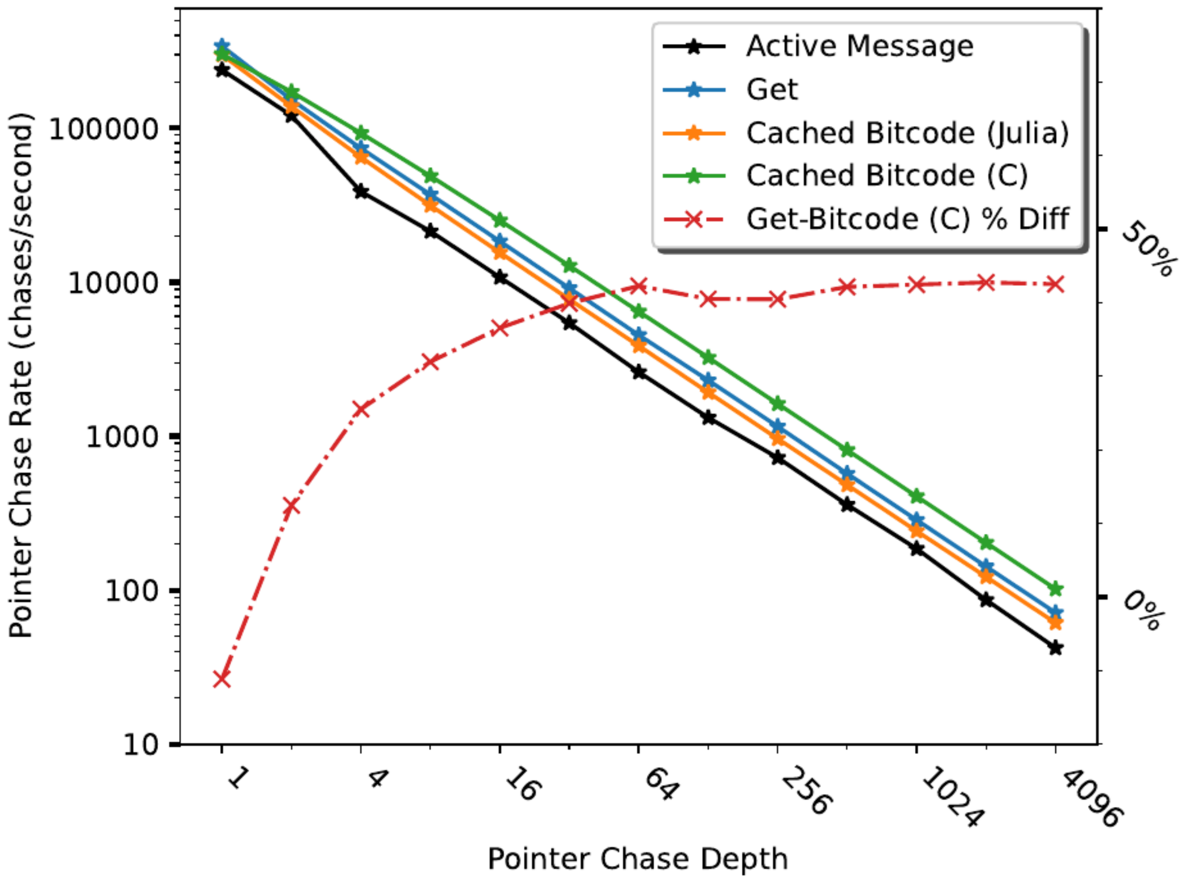


Ookami 64-Server **C/C++**
(A64FX Client and Servers)

Results: Julia

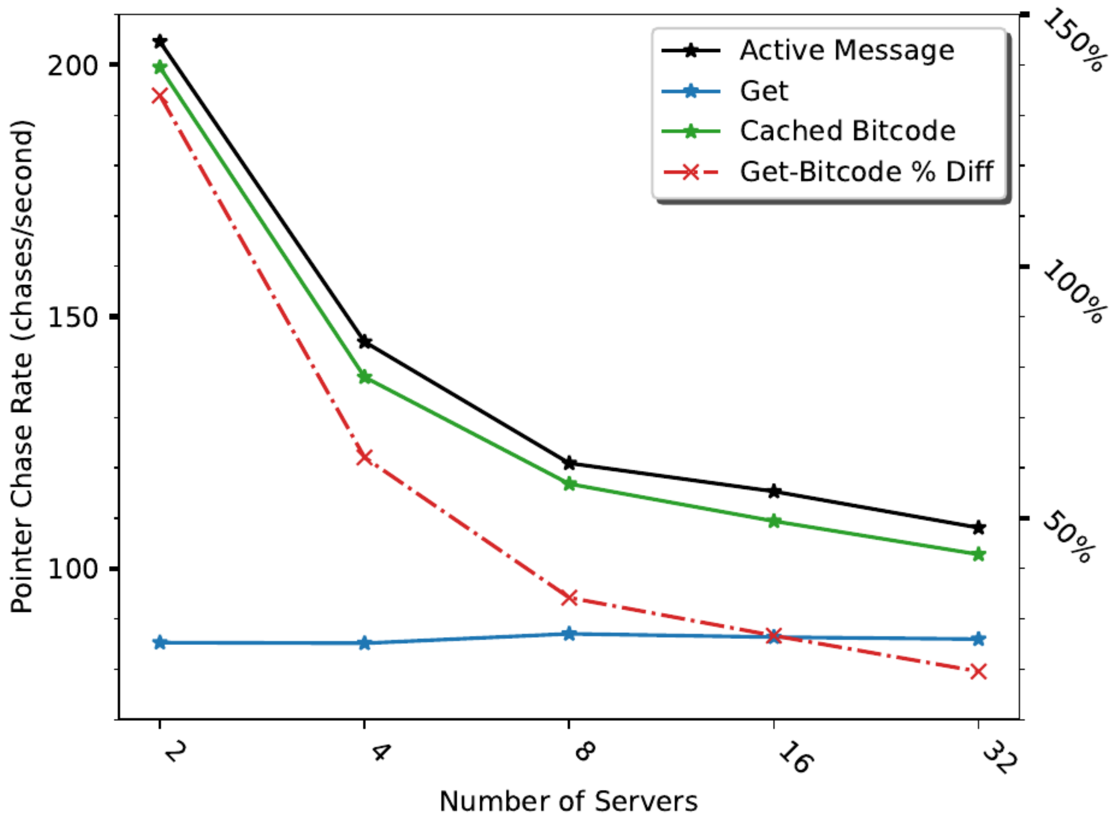


Thor 32-Server **C/C++**
(Xeon Client and BF2 Servers)

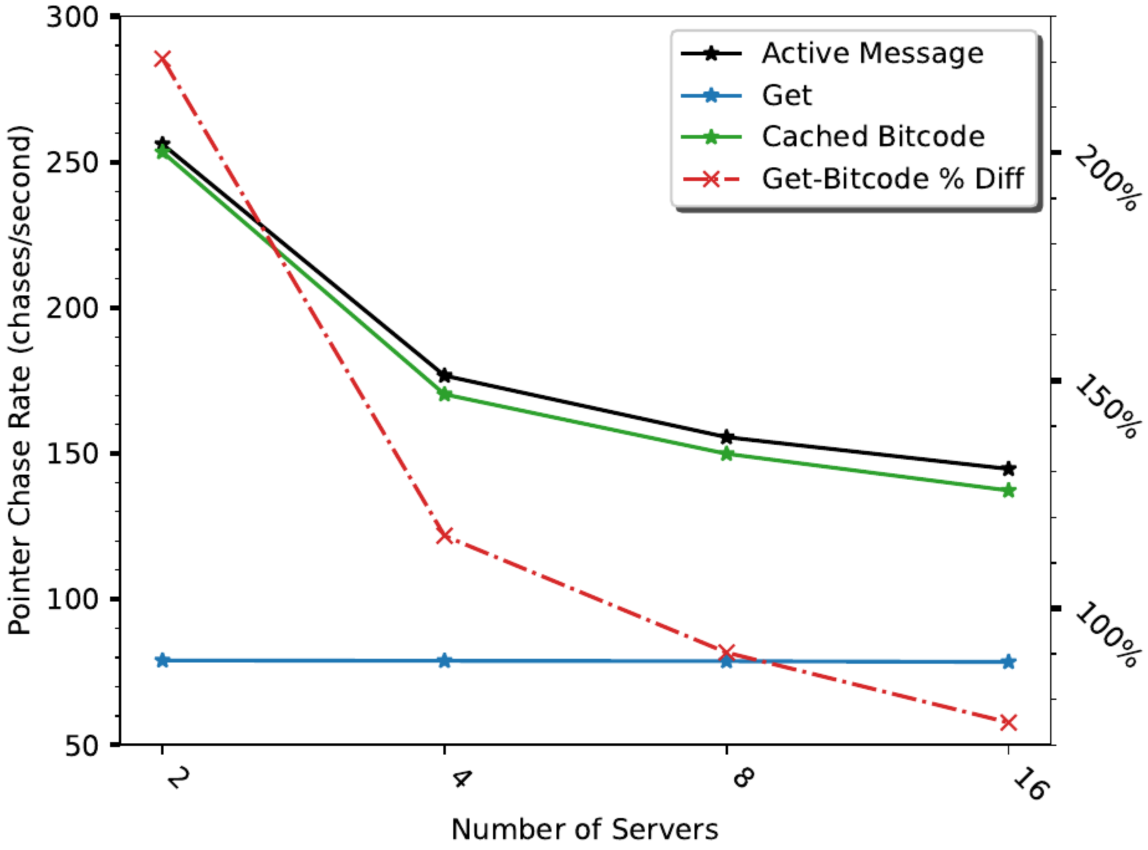


Thor 32-Server **Julia**
(Xeon Client and BF2 Servers)

Results: Xeon-BF2 vs Xeon

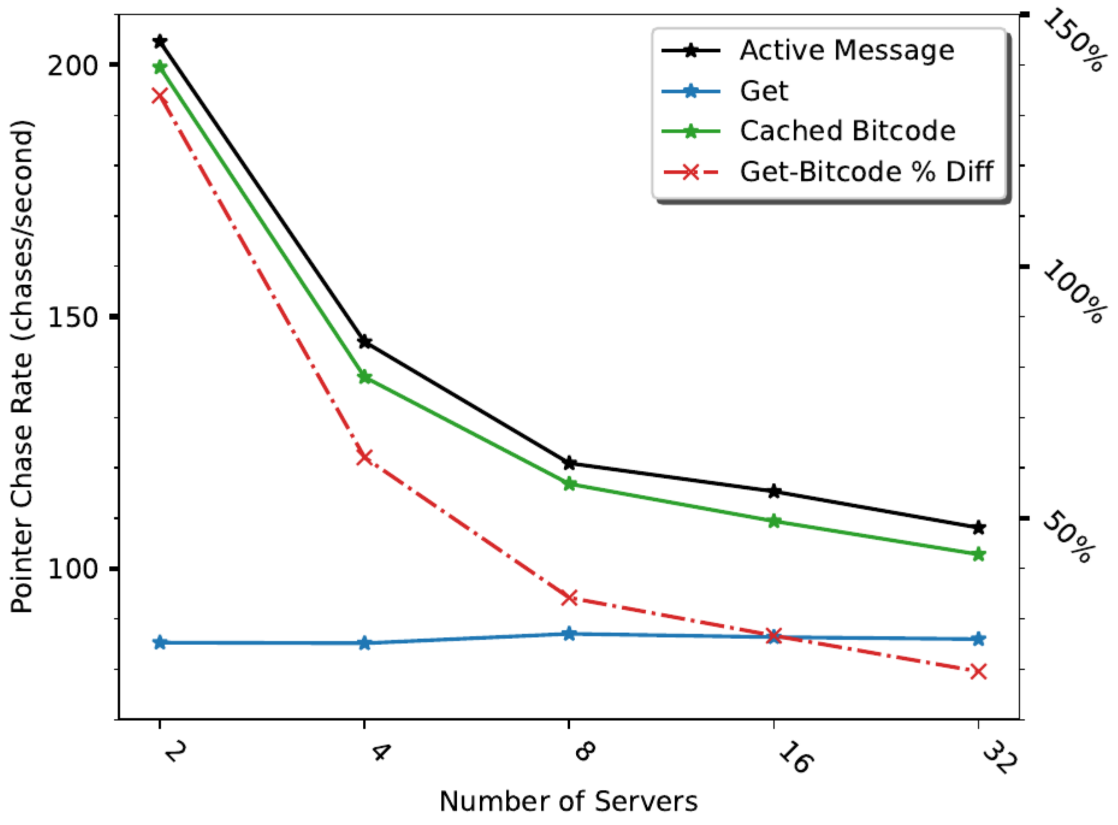


Thor 4096-Depth **C/C++**
(Xeon Client and BF2 Servers)

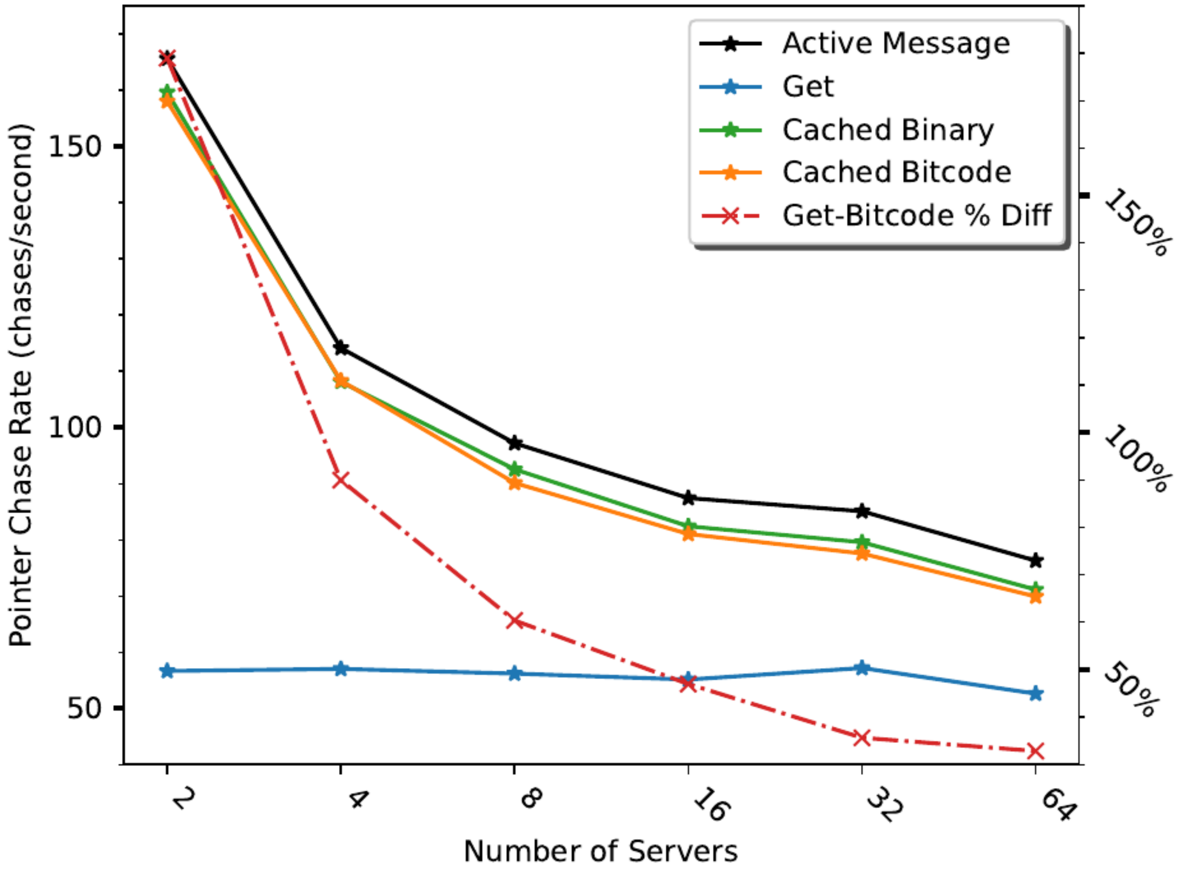


Thor 4096-Depth **C/C++**
(Xeon Client and Servers)

Results: Xeon-BF2 vs A64FX

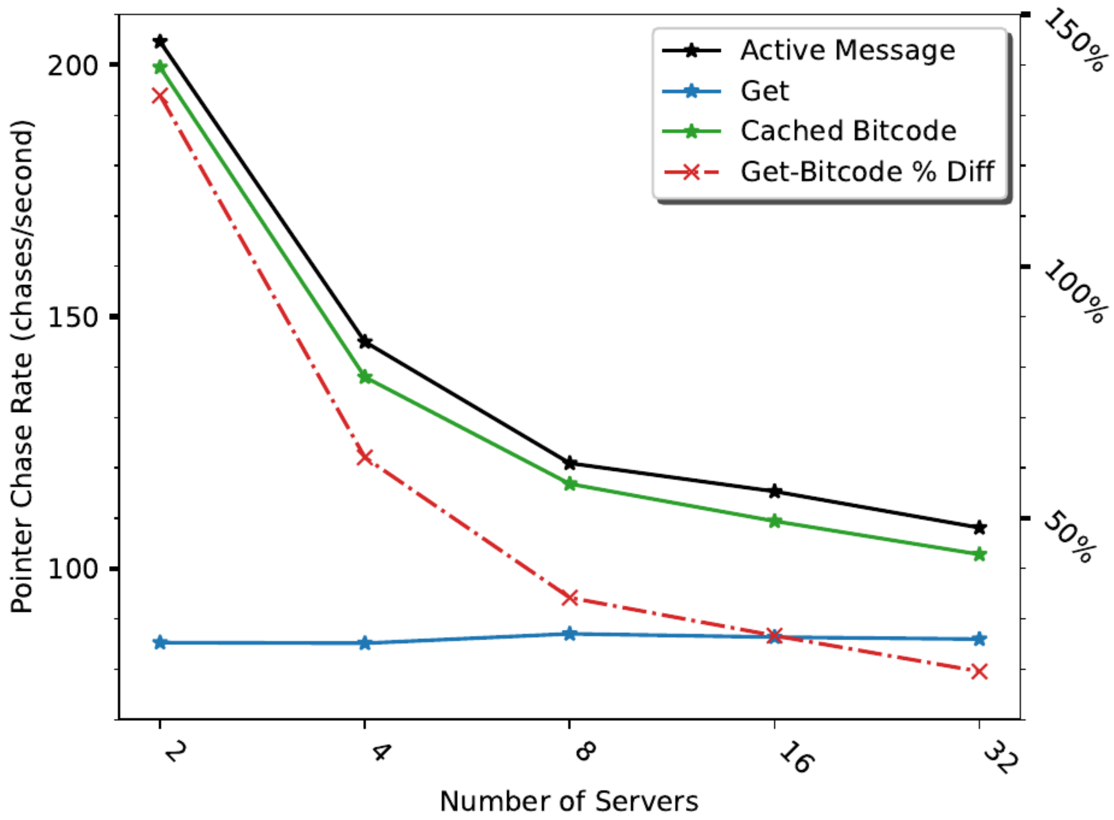


Thor 4096-Depth **C/C++**
(Xeon Client and BF2 Servers)

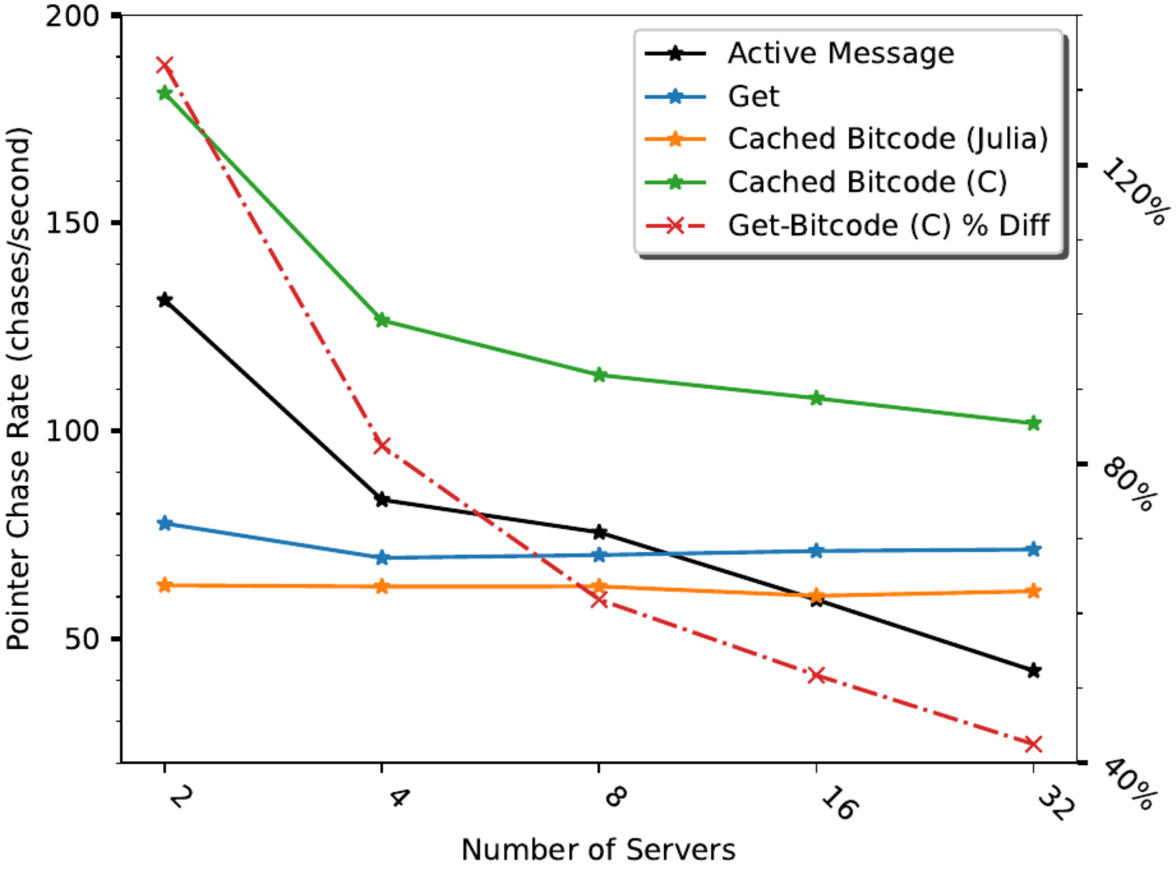


Ookami 4096-Depth **C/C++**
(A64FX Client and Servers)

Results: Julia



Thor 4096-Depth **C/C++**
(Xeon Client and BF2 Servers)



Thor 4096-Depth **Julia**
(Xeon Client and BF2 Servers)

TSI Overhead breakdown (Thor BF2)

Stage	Active Message	JIT compiled Bitcode	Cached Bitcode
Lookup+Exec	0.01 µs	0.04 µs	0.01 µs
JIT	N/A	(4.50 ms)	N/A
Transmission	1.87 µs	3.45 µs	1.85 µs
Total	1.88 µs	3.49 µs	1.86 µs

Method	Latency	Speedup	Message Rate	Speedup
Active Message	1.88 µs	0.86%	974,000 msg/sec	34.60%
Cached Bitcode	1.87 µs		1,311,000 msg/sec	
Uncached Bitcode	3.49 µs	87.73%	417,300 msg/sec	214.16%
Cached Bitcode	1.87 µs		1,311,000 msg/sec	

Conclusion / Next steps

- Bitcode propagation over the network
- Fast programming of network attached heterogeneous resources
  - Can we extend this to AWS Lambda/Serverless like architectures

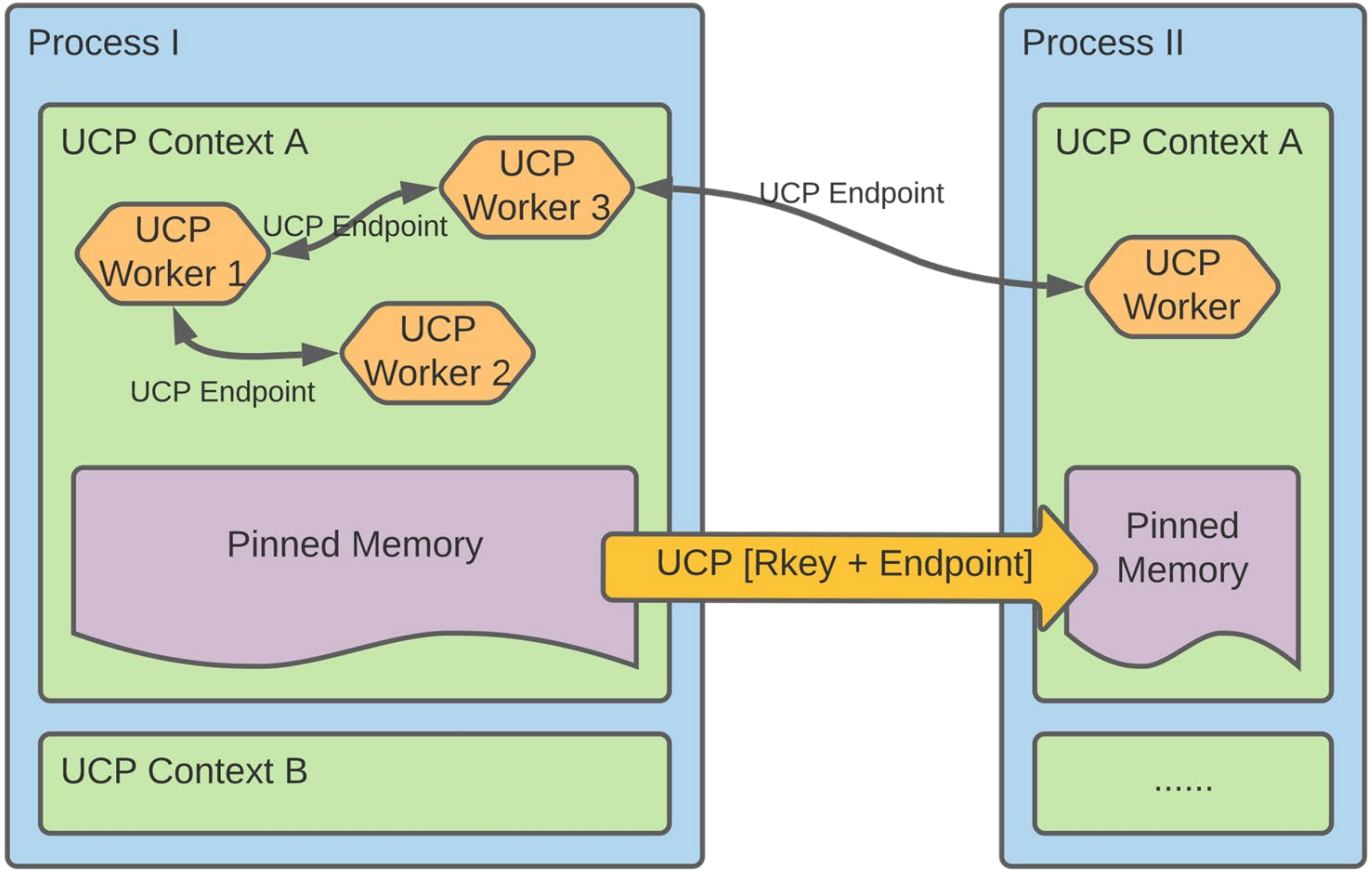
- Security: WASM/eBPF
- Initial prototype inside UCX: Next step separate library
- Explore range of choices:
  - Pre-opt for computational intensive
  - PIC object-files for latency sensitive work.

Improved static/cross compilation for Julia

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה

Bonus slides

More on UCX


Process I
UCP Context A
UCP Worker 3
UCP Endpoint
UCP Worker 1
UCP Endpoint
UCP Worker 2
UCP Endpoint
Pinned Memory
UCP Context B
UCP Endpoint
UCP [Rkey + Endpoint]
Process II
UCP Context A
UCP Worker
Pinned Memory
......

arm