Quick Start: Beethoven CLI

ISCA 2026 Tutorial

Duke

Quick Start - Prerequisites

- Let's build the Beethoven project in 5 minutes!

- Beethoven supports and has been tested on:
  - Linux (x86-64) & macOS (arm64)
  - HomeBrew is required for Mac: https://brew.sh/

- Windows users may use WSL or Docker Desktop
  - https://learn.microsoft.com/en-us/windows/wsl/install

Duke

Quick Start - Installing Dependencies

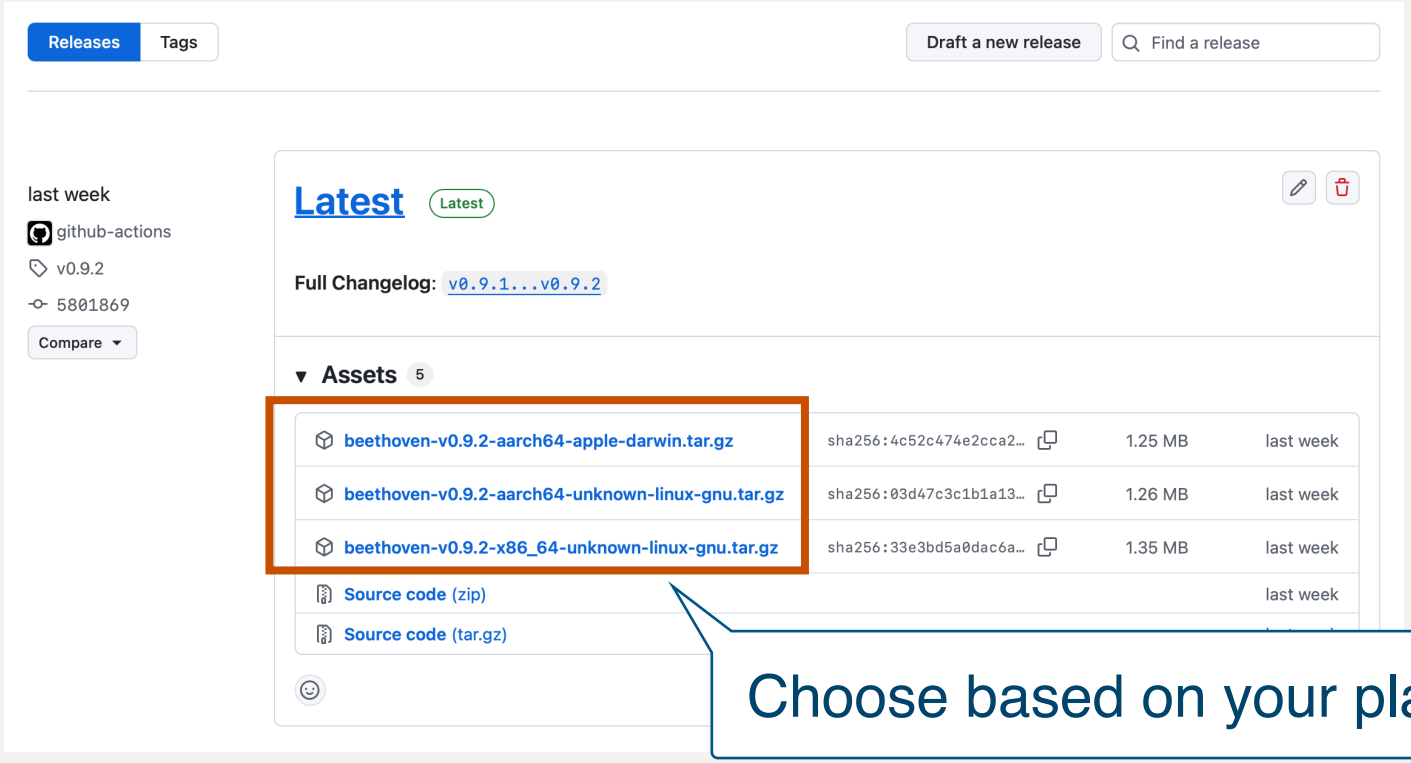
- Scala is required for Beethoven.
- Recommended way - Install via SDK manager:

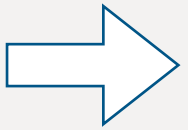
- Next, we'll guide you through running your first project in 3 simple steps!

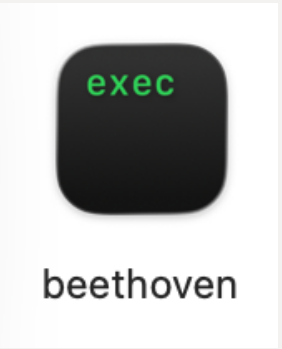
Duke

Download pre-compiled Beethoven-CLI tool

- Beethoven-cli is an all-in-one tool for project management.
- https://github.com/Composer-Team/Beethoven-Software/releases







- Tip: If you prefer building your own, you'll need to clone this repo and run `cargo build` inside ./cli.

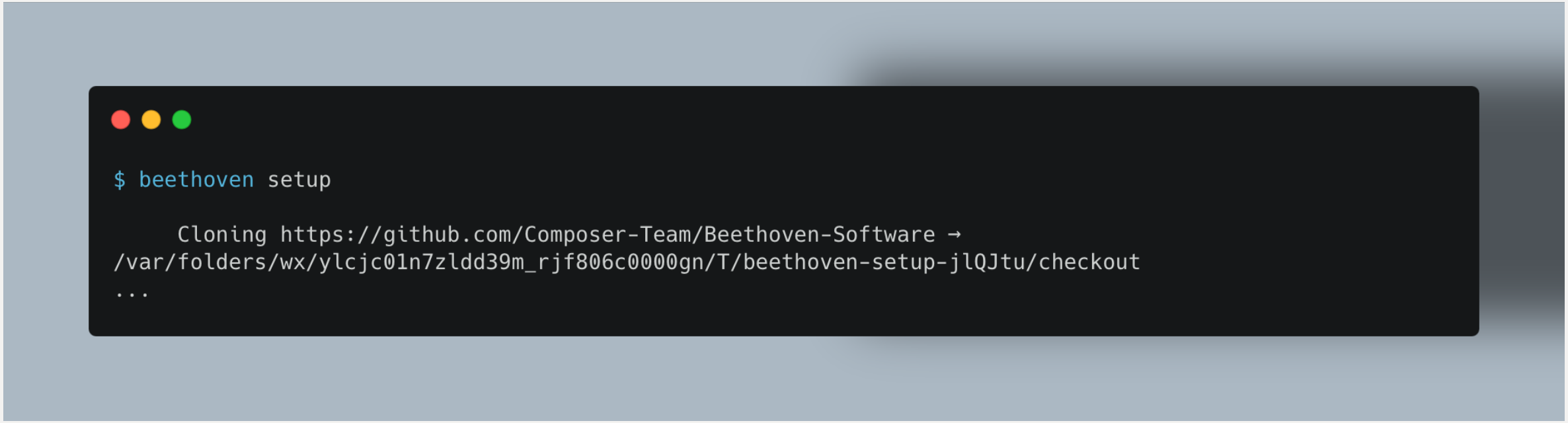
Duke

Try it in your Terminal!

- You are ready to go if you see the screenshot on the right.
- Use `--help` to show docs for each command.
- Mac users may have to allow execution in System Settings-> Privacy & Security

Duke

Auto-setup using Beethoven-CLI

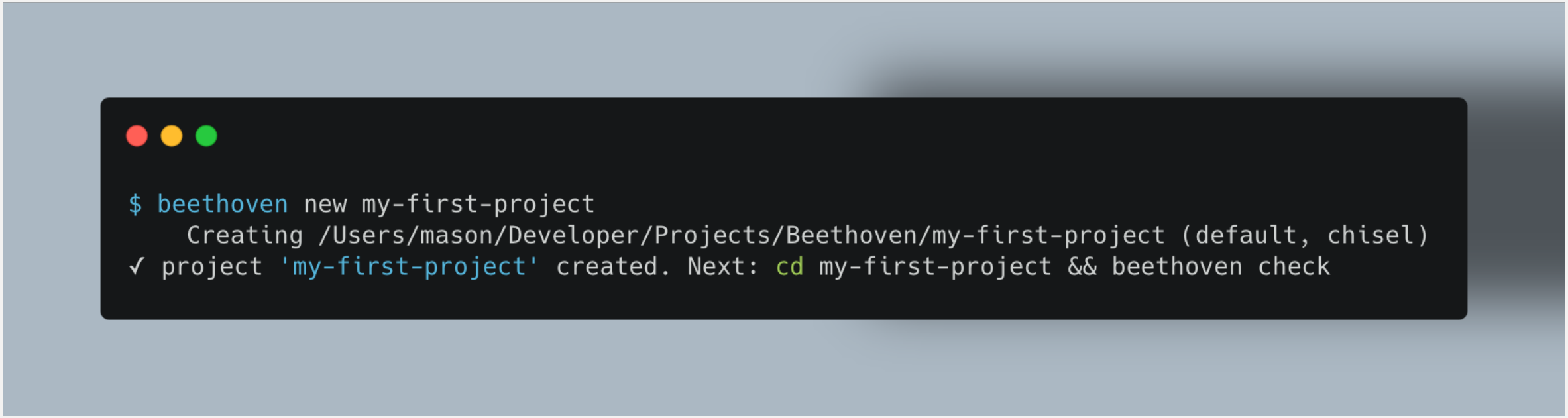


- First time bootstrap - downloading necessary components for Beethoven to work.

Duke

Create, build, simulate, in 3 commands!

- Beethoven will create a minimal working project under a new folder.
- Feel free to explore what's inside - we'll cover them later as well.

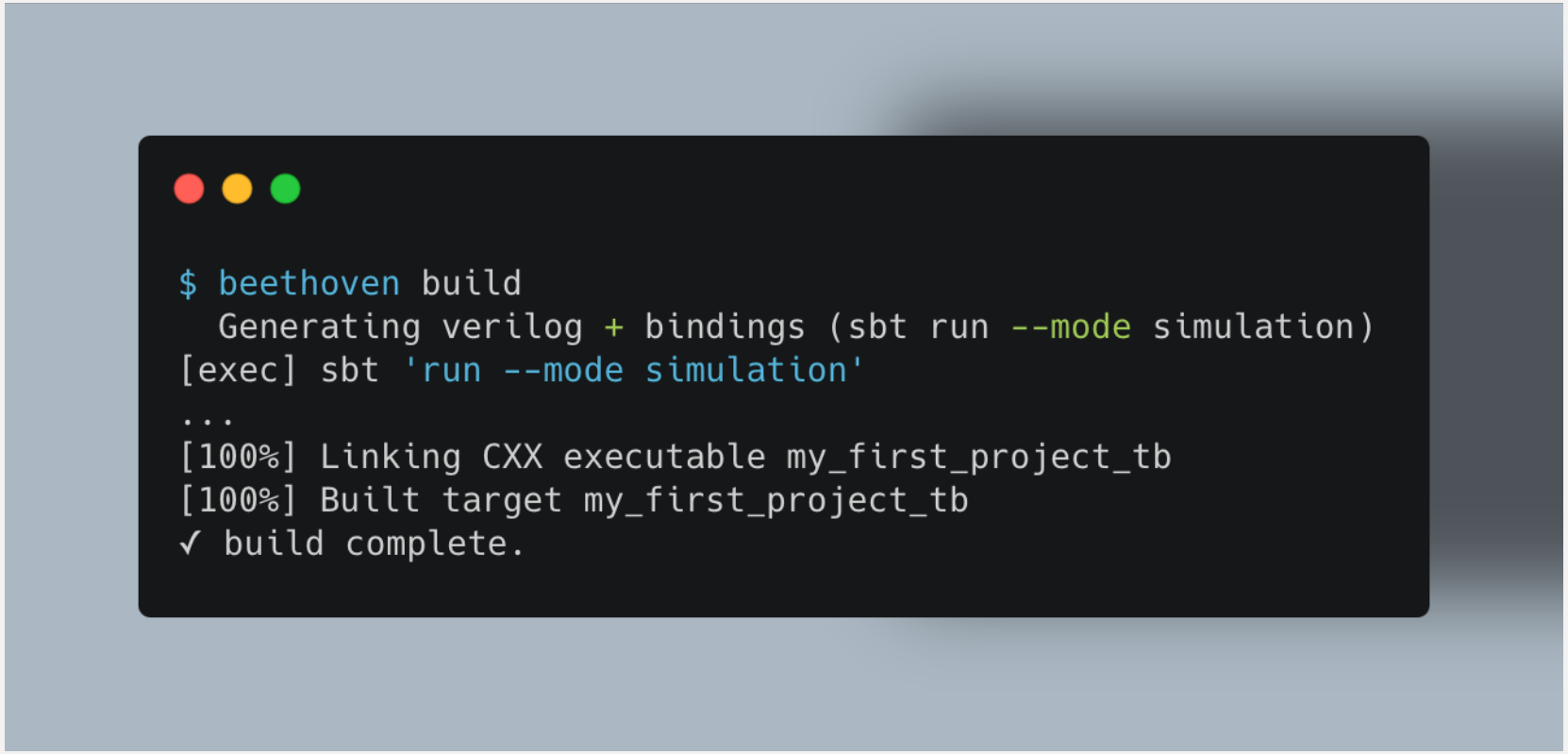


- Tip: Verilog fans may add `--verilog` argument to create a new Verilog (instead of Chisel) project.

Duke

Create, build, simulate, in 3 commands!

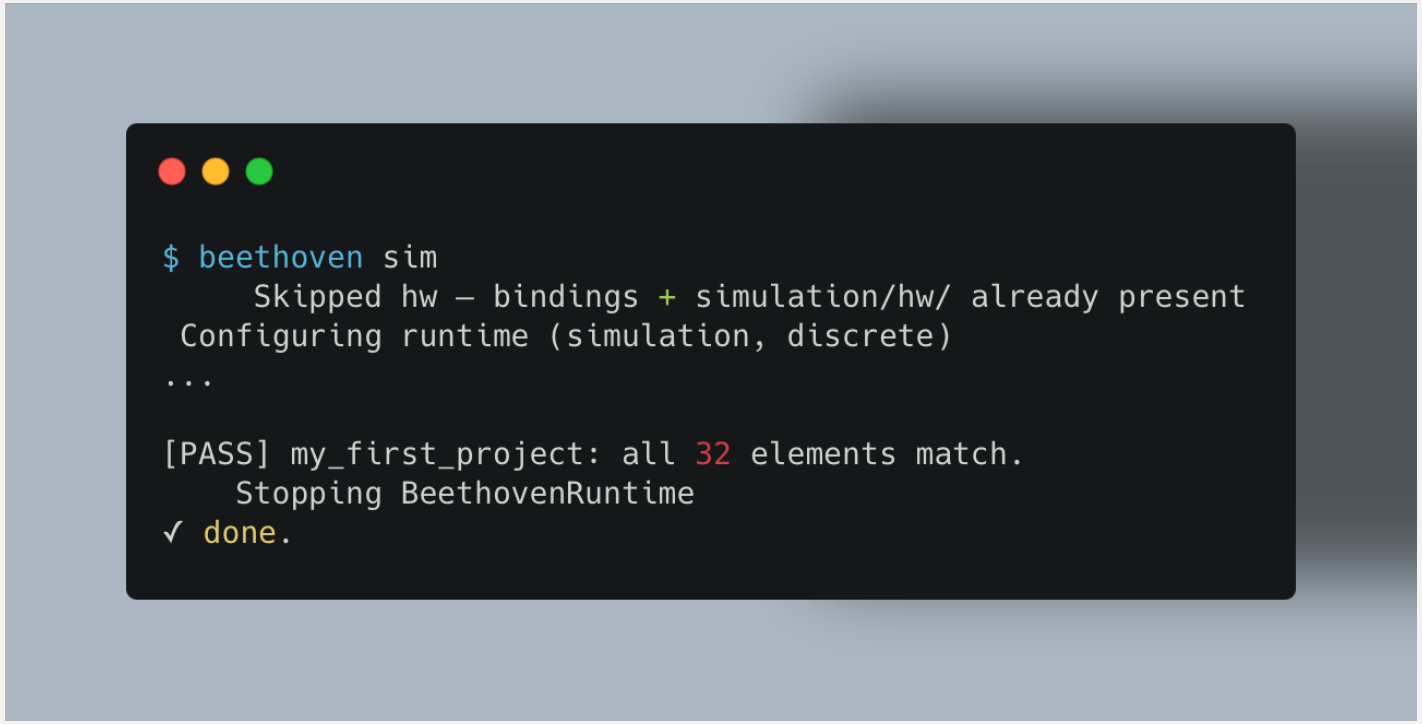
- cd into my-first-project and run `beethoven build` to build both hardware and software.



Duke

Create, build, simulate, in 3 commands!

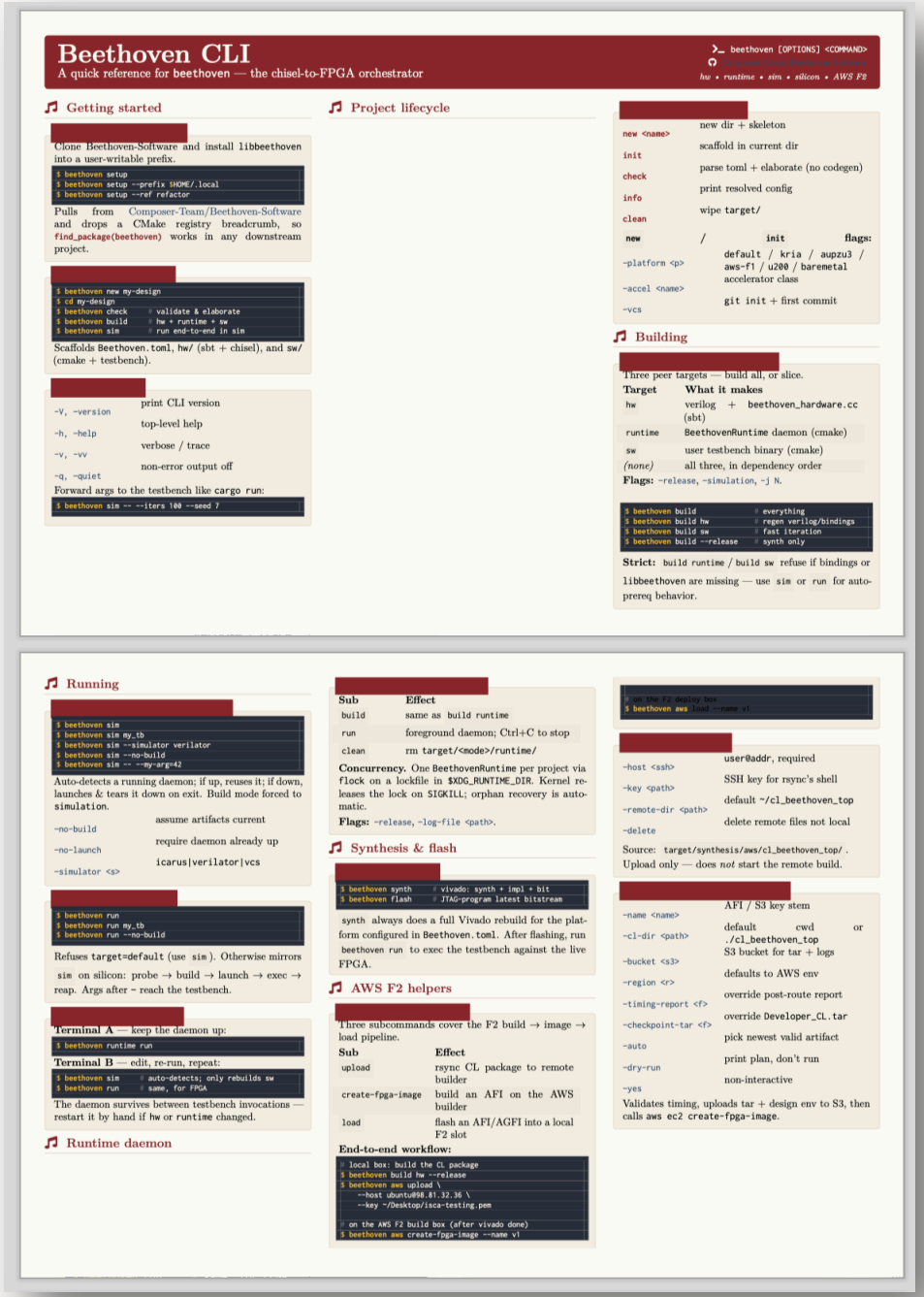
- Launch HW simulation and software with one command:



- Congrats! You have created, built, and simulated your very first Beethoven project.
- Tip: You may find the generated waveform at: my-first-project/target/simulation/dump.vcd

Duke

More Info:
CLI Cheatsheet



Beethoven CLI

A quick reference for **beethoven** — the chisel-to-FPGA orchestrator

`beethoven [OPTIONS] <COMMAND>`

Version Your Hardware and Software

hw • runtime • sim • silicon • AWS F2

Getting started

Clone Beethoven-Software and install `libbeethoven` into a user-writable prefix.

```
$ beethoven setup
$ beethoven setup --prefix $HOME/.local
$ beethoven setup --ref refactor
```

Pulls from `Composer-Team/Beethoven-Software` and drops a CMake registry breadcrumb, so `find_package(beethoven)` works in any downstream project.

```
$ beethoven new my-design
$ cd my-design
$ beethoven check     # validate & elaborate
$ beethoven build     # hw + runtime + sw
$ beethoven sim       # run end-to-end in sim
```

Scaffolds `Beethoven.toml`, `hw/` (sbt + chisel), and `sw/` (cmake + testbench).

`-V, --version`	print CLI version
`-h, --help`	top-level help
`-v, -vv`	verbose / trace
`-q, --quiet`	non-error output off

Forward args to the testbench like `cargo run`:

```
$ beethoven sim -- --iters 100 --seed 7
```

Project lifecycle

`new <name>`	new dir + skeleton
`init`	scaffold in current dir
`check`	parse toml + elaborate (no codegen)
`info`	print resolved config
`clean`	wipe `target/`

`new` / **`init`** **flags:**

`-platform <p>`	`default` / `kria` / `aupzu3` / `aws-f1` / `u200` / `baremetal` accelerator class
`-accel <name>`	
`-vcs`	`git init` + first commit

Building

Three peer targets — build all, or slice.

Target	What it makes
`hw`	verilog + `beethoven_hardware.cc` (sbt)
`runtime`	`BeethovenRuntime` daemon (cmake)
`sw`	user testbench binary (cmake)
(none)	all three, in dependency order

Flags: `-release`, `-simulation`, `-j N`.

```
$ beethoven build             # everything
$ beethoven build hw          # regen verilog/bindings
$ beethoven build sw          # fast iteration
$ beethoven build --release   # synth only
```

Strict: `build runtime` / `build sw` refuse if bindings or `libbeethoven` are missing — use `sim` or `run` for auto-prereq behavior.

Running

```
$ beethoven sim
$ beethoven sim my_tb
$ beethoven sim --simulator verilator
$ beethoven sim --no-build
$ beethoven sim -- --my-arg=42
```

Auto-detects a running daemon; if up, reuses it; if down, launches & tears it down on exit. Build mode forced to `simulation`.

`-no-build`	assume artifacts current
`-no-launch`	require daemon already up
`-simulator <s>`	`icarus|verilator|vcs`

```
$ beethoven run
$ beethoven run my_tb
$ beethoven run --no-build
```

Refuses `target=default` (use `sim`). Otherwise mirrors `sim` on silicon: probe → build → launch → exec → reap. Args after `--` reach the testbench.

Terminal A — keep the daemon up:

```
$ beethoven runtime run
```

Terminal B — edit, re-run, repeat:

```
$ beethoven sim     # auto-detects; only rebuilds sw
$ beethoven run     # same, for FPGA
```

The daemon survives between testbench invocations — restart it by hand if `hw` or `runtime` changed.

Runtime daemon

Sub	Effect
`build`	same as `build runtime`
`run`	foreground daemon; Ctrl+C to stop
`clean`	rm `target/<mode>/runtime/`

Concurrency. One `BeethovenRuntime` per project via `flock` on a lockfile in `$XDG_RUNTIME_DIR`. Kernel releases the lock on `SIGKILL`; orphan recovery is automatic.

Flags: `-release`, `-log-file <path>`.

Synthesis & flash

```
$ beethoven synth     # vivado: synth + impl + bit
$ beethoven flash     # JTAG-program latest bitstream
```

`synth` always does a full Vivado rebuild for the platform configured in `Beethoven.toml`. After flashing, run `beethoven run` to exec the testbench against the live FPGA.

AWS F2 helpers

Three subcommands cover the F2 build → image → load pipeline.

Sub	Effect
`upload`	rsync CL package to remote builder
`create-fpga-image`	build an AFI on the AWS builder
`load`	flash an AFI/AGFI into a local F2 slot

End-to-end workflow:

```
# local box: build the CL package
$ beethoven build hw --release
$ beethoven aws upload \
    --host ubuntu@98.81.32.36 \
    --key ~/Desktop/isca-testing.pem

# on the AWS F2 build box (after vivado done)
$ beethoven aws create-fpga-image --name v1

# on the F2 deploy box
$ beethoven aws load --name v1
```

`-host <ssh>`	`user@addr`, required
`-key <path>`	SSH key for rsync's shell
`-remote-dir <path>`	default `~/cl_beethoven_top`
`-delete`	delete remote files not local

Source: `target/synthesis/aws/cl_beethoven_top/`. Upload only — does *not* start the remote build.

`-name <name>`	AFI / S3 key stem
`-cl-dir <path>`	default cwd or `./cl_beethoven_top`
`-bucket <s3>`	S3 bucket for tar + logs
`-region <r>`	defaults to AWS env
`-timing-report <f>`	override post-route report
`-checkpoint-tar <f>`	override `Developer_CL.tar`
`-auto`	pick newest valid artifact
`-dry-run`	print plan, don't run
`-yes`	non-interactive

Validates timing, uploads tar + design env to S3, then calls `aws ec2 create-fpga-image`.

Duke

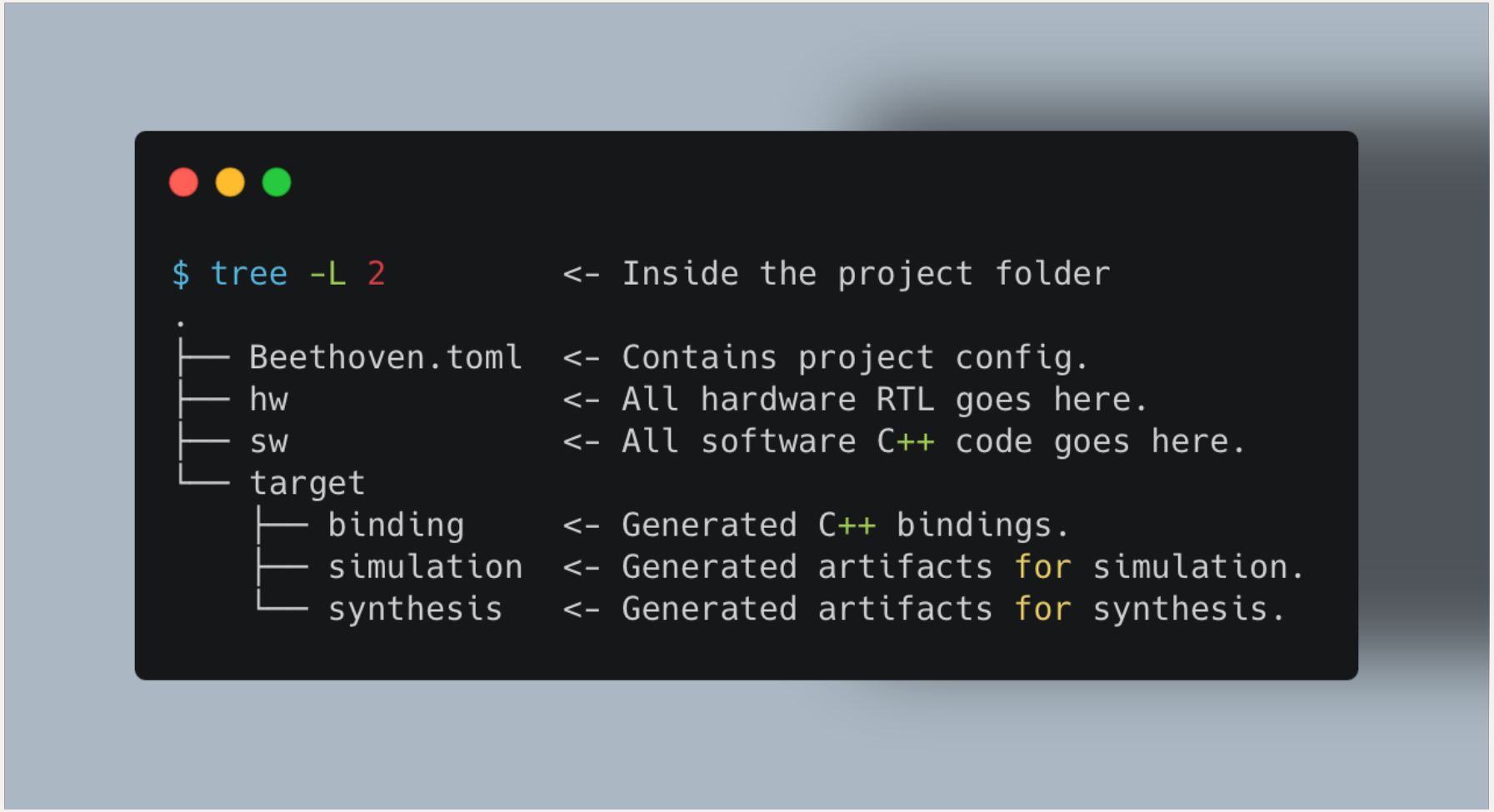
Beethoven Project Structure

ISCA 2026 Tutorial

Duke

Project Structure

- Each Beethoven project lives in its own folder. We'll cover hardware and software later.



Duke

Beethoven.toml - the project config file

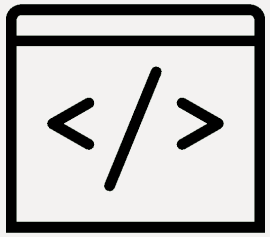
```toml
# Beethoven project manifest
[project]
name = "my_first_project"

## Beethoven-Hardware dependency.
[hardware.beethoven-hardware]
version = "latest.integration"
# version = "0.1.7-dev12"
# path = "../Beethoven-Hardware"

[platform]
## Deployment target. One of:
##   default | kria | kria2 | aupzu3 | aws-f1 | aws-f2 | u200
##
target = "default"
```

Project name

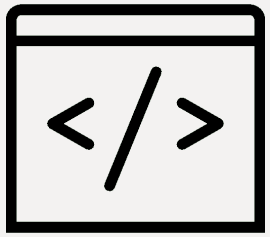
Beethoven-Hardware location:
Keep it default.

Target hardware platform:
Default: simulation-only for quick prototyping

Behind the Scene: What components does Beethoven have?



Beethoven-Hardware - hardware and binding generator
https://github.com/Composer-Team/Beethoven-Hardware
Auto-installed by CLI tool at `~/.ivy2/local/`

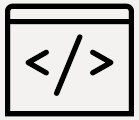


Beethoven-Software - CLI tool, software runtime and dependencies
https://github.com/Composer-Team/Beethoven-Software
Auto-installed by CLI tool at `~/.local/share/beethoven/`

Duke

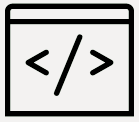
Beethoven-Zoo: Curated examples to explore!

- Link: https://github.com/Composer-Team/Beethoven-Zoo
- Please clone it to a local directory

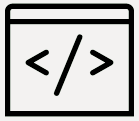


GameBoy Emulator: hardware-accelerated GameBoy Color gaming.

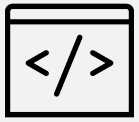
Dot Product: a simple example we gonna walkthrough



SHAKE256: Crypto accelerator in Verilog.



A3: Multi-core attention accelerator.



Systolic Array: a simple systolic-array accelerator as an exercise

Emulator screenshot of Super Mario Bros

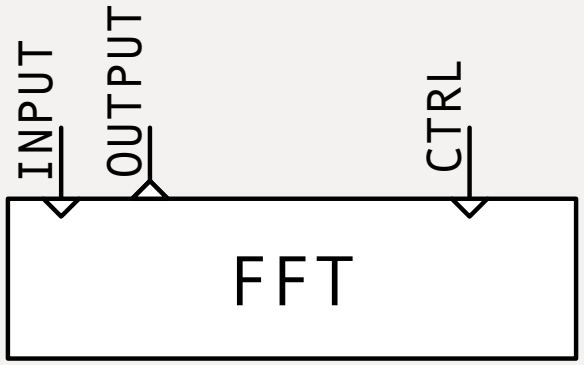
Duke

Motivation

Why Beethoven?

System Overview

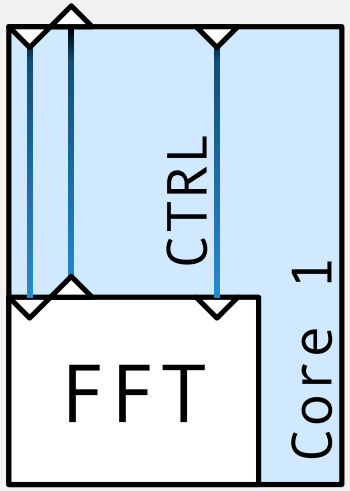
Start with your core algorithm



Duke

System Overview

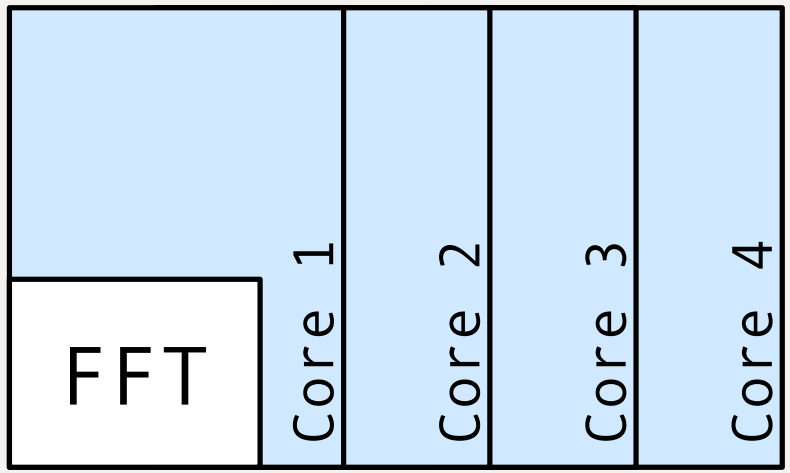
Start with your core algorithm



Duke

System Overview

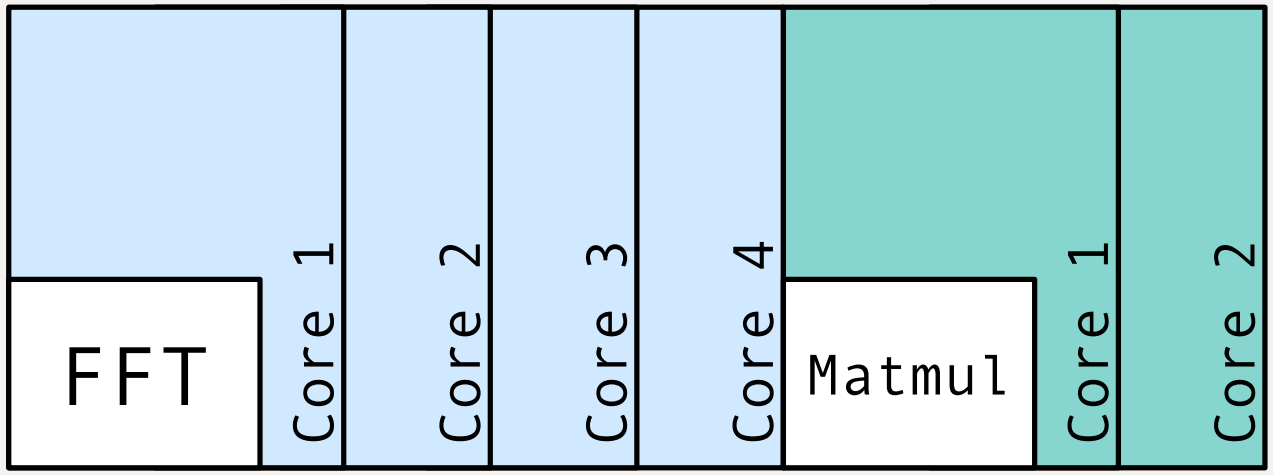
Data-parallelism through multiple cores



Duke

System Overview

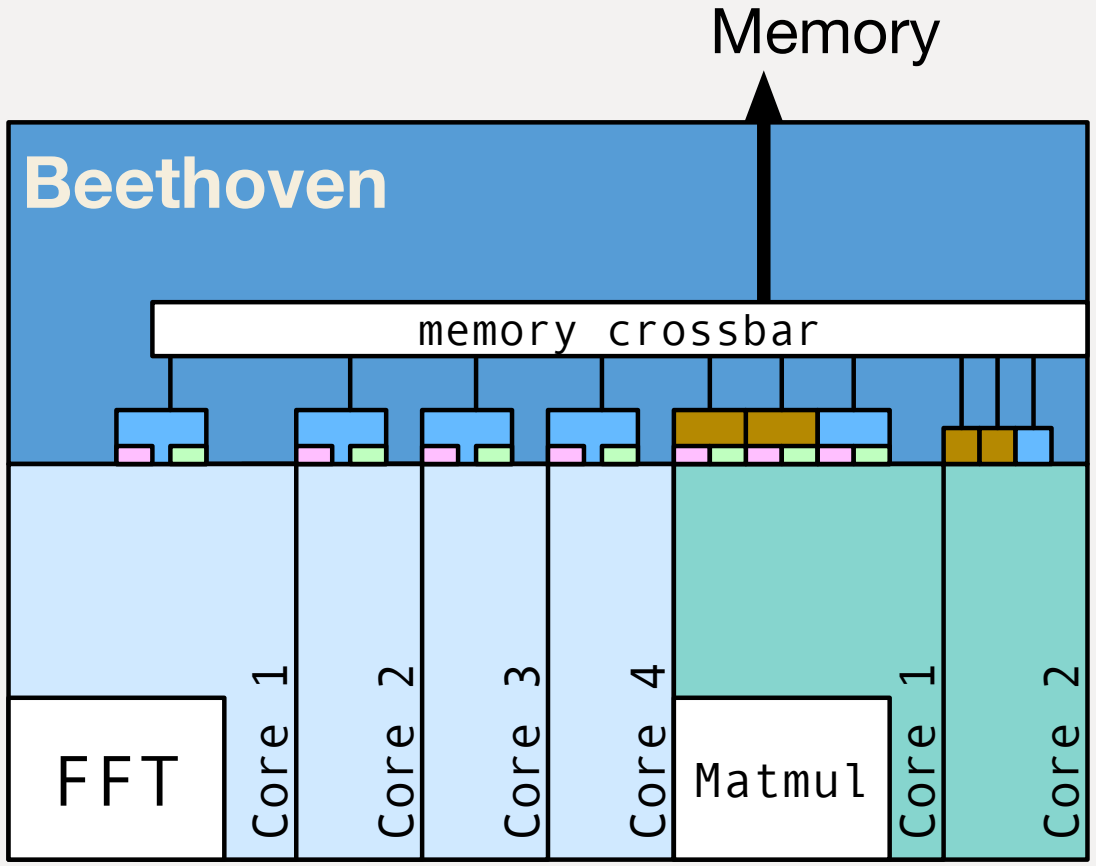
*Task-Parallelism through different **types** of cores*



Duke

System Overview

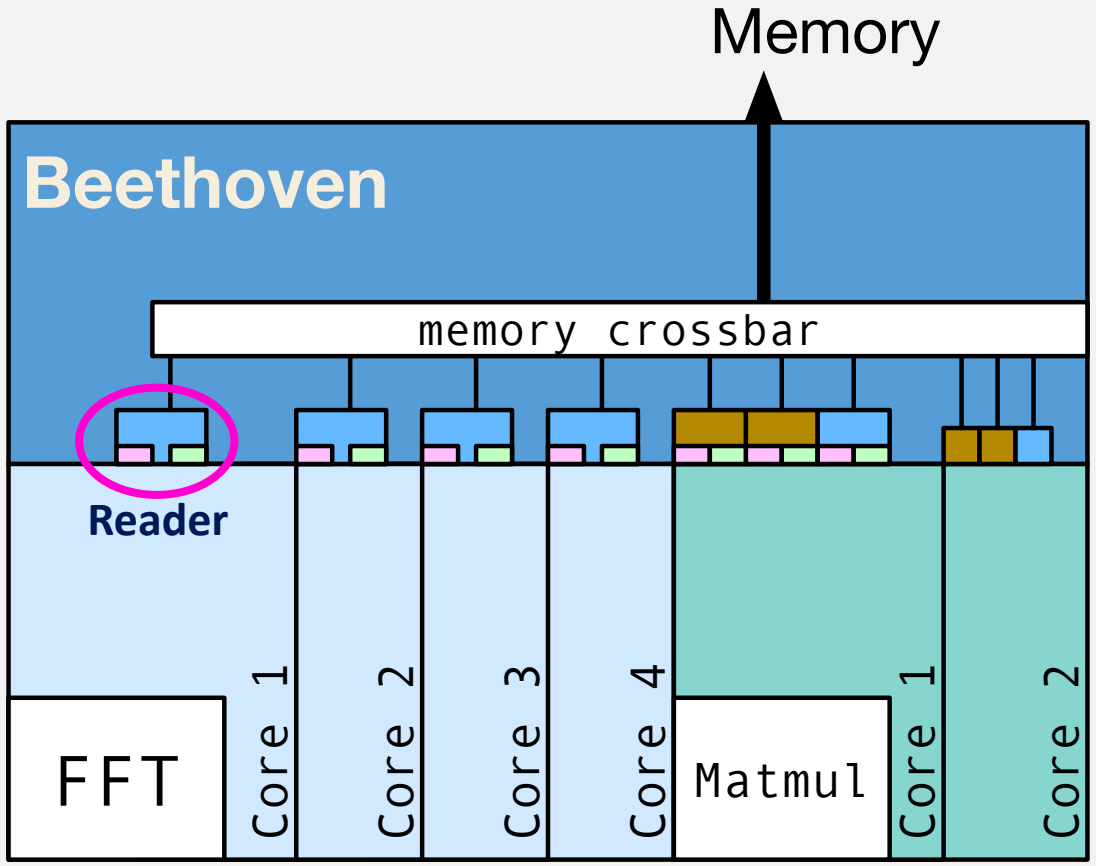
Address heterogenous memory needs through customizable interfaces



Duke

System Overview

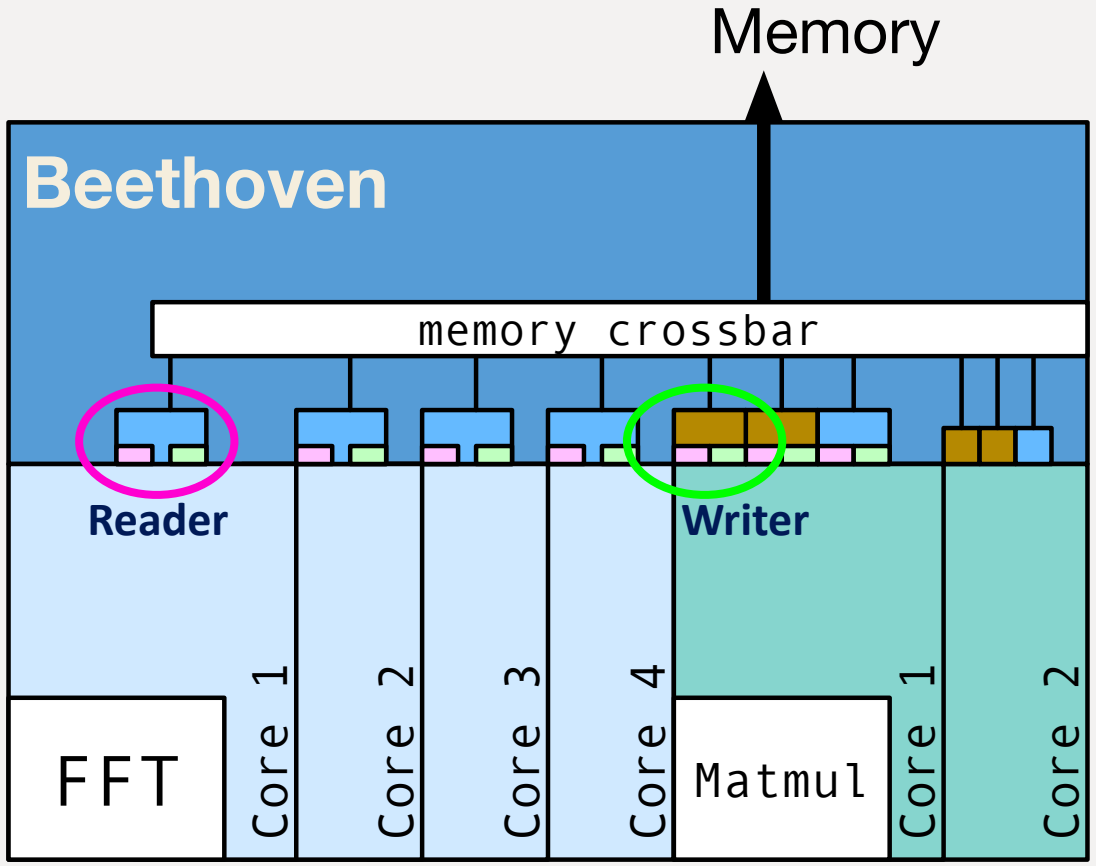
Address heterogenous memory needs through customizable interfaces



Duke

System Overview

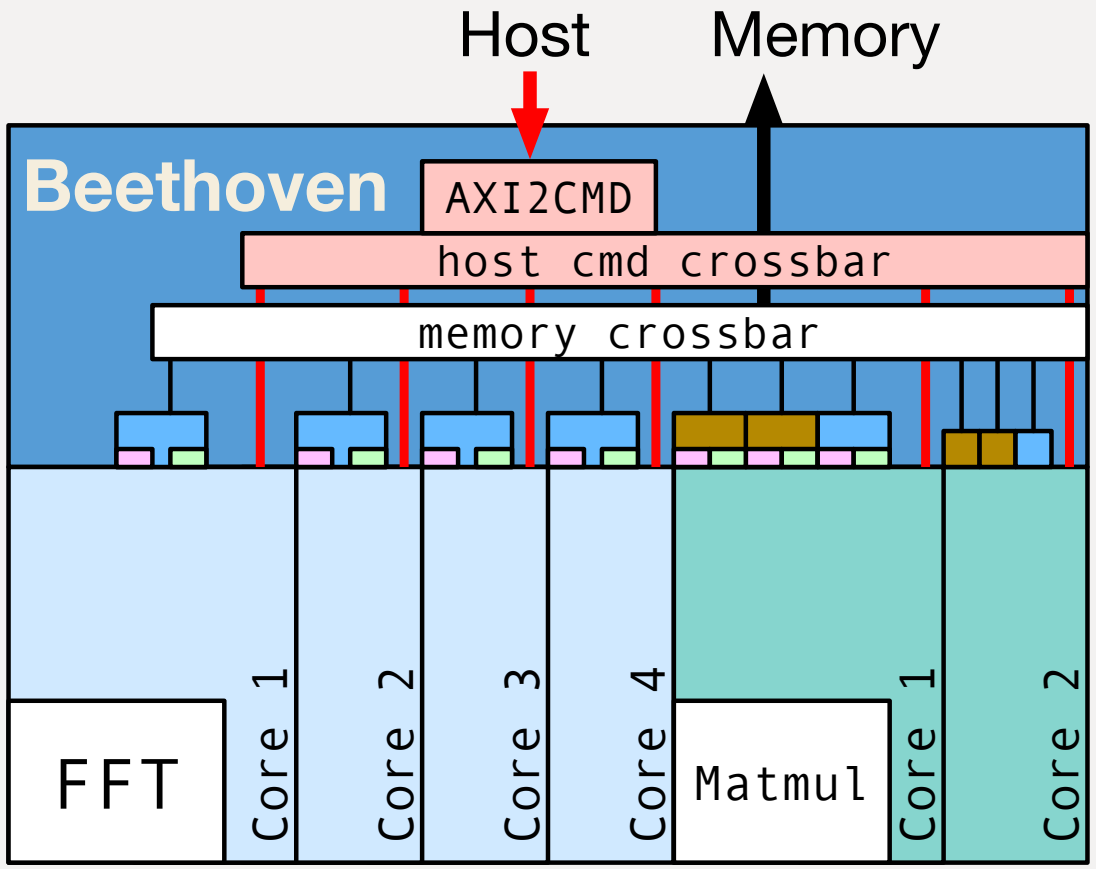
Address heterogenous memory needs through customizable interfaces



Duke

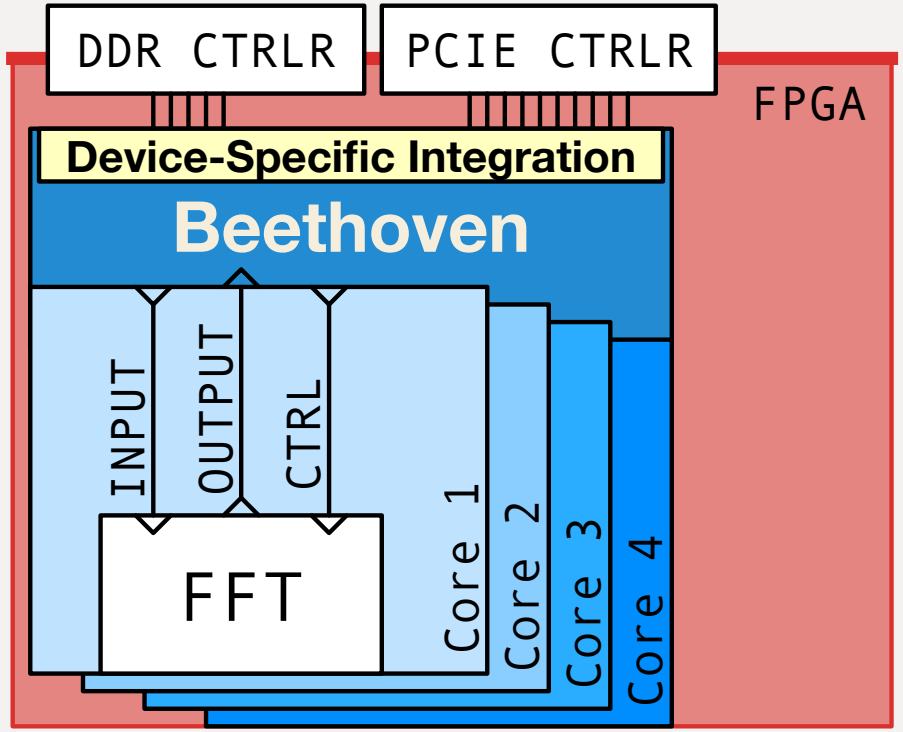
System Overview

Host-to-Accelerator Kernel Launch/HW Communication



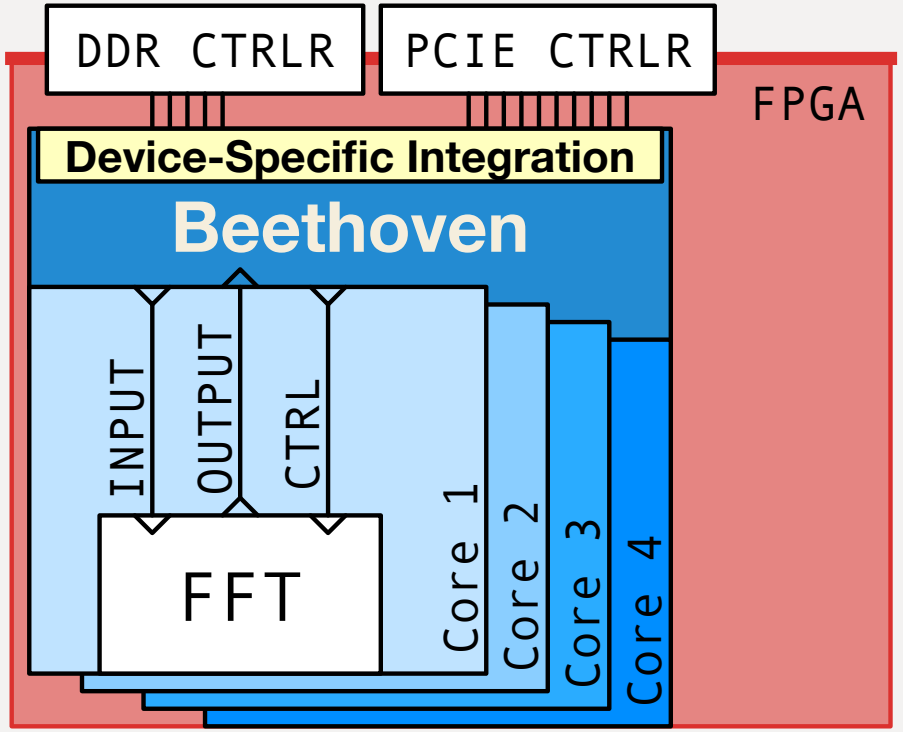
Duke

Portability through Separation of Concerns



Portability through Separation of Concerns

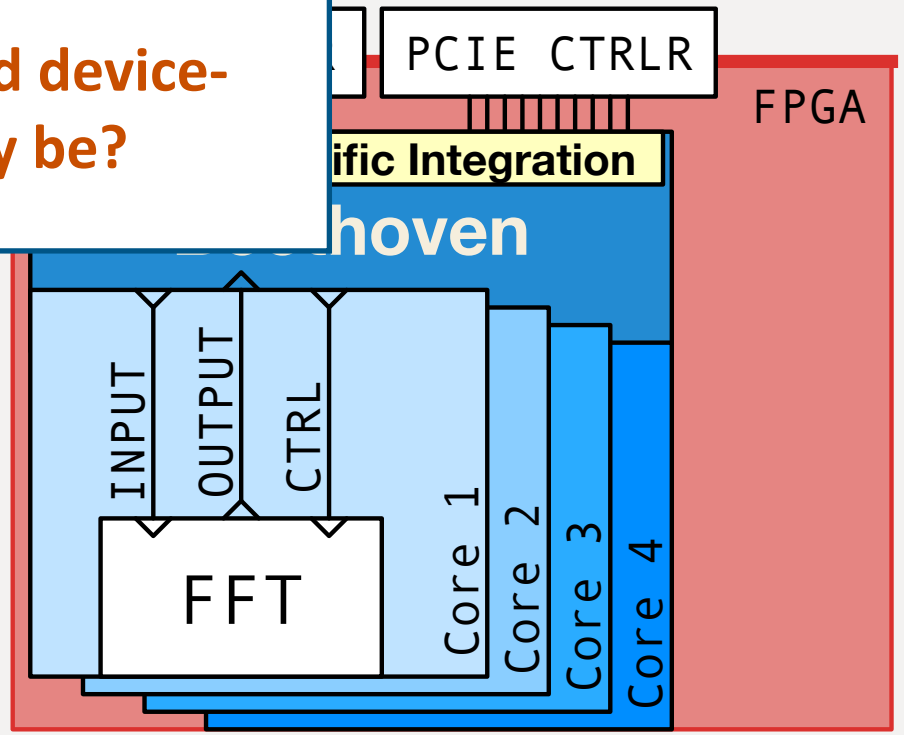
- **Device engineers** develop re-usable platform integrations



Portability through Separation of Concerns

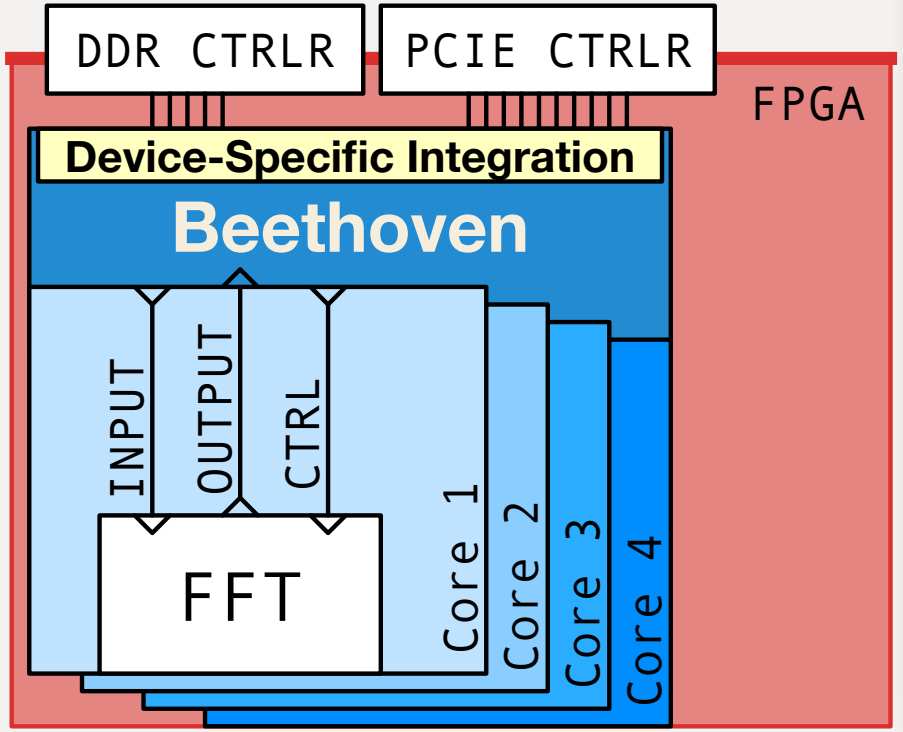
- **Device engineers** develop re-usable platform integrations

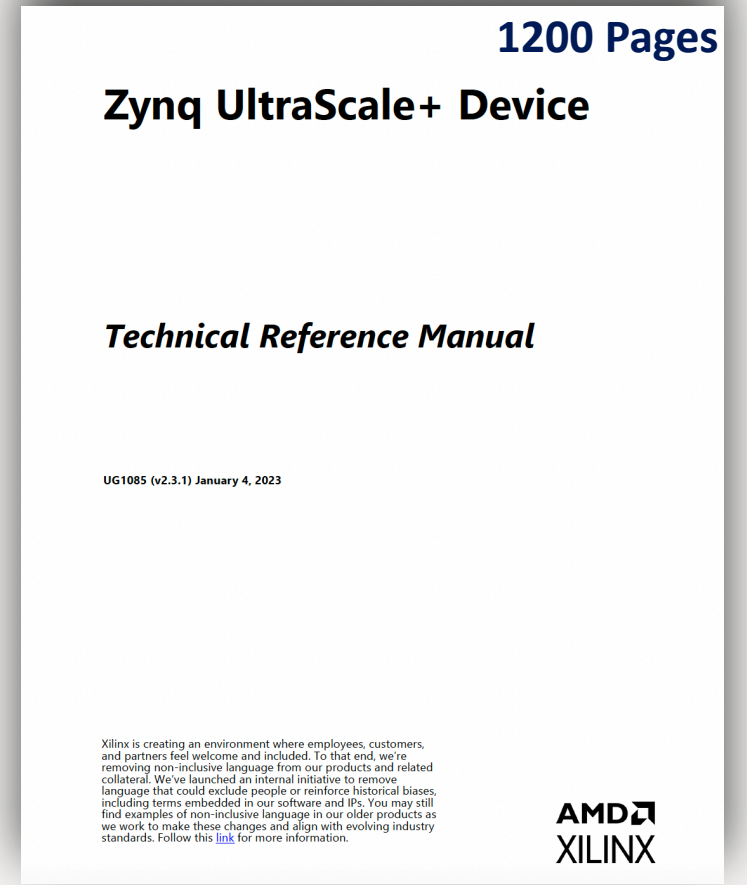
How complicated could device-integration actually be?

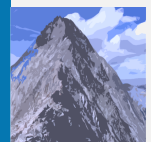


Portability through Separation of Concerns

- **Device engineers** develop re-usable platform integrations

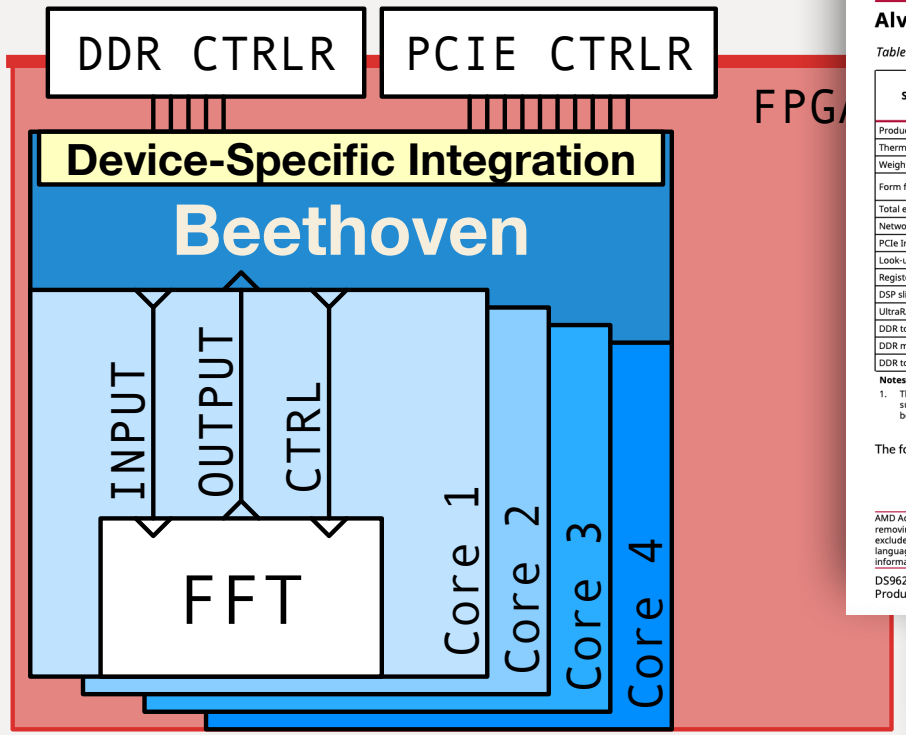


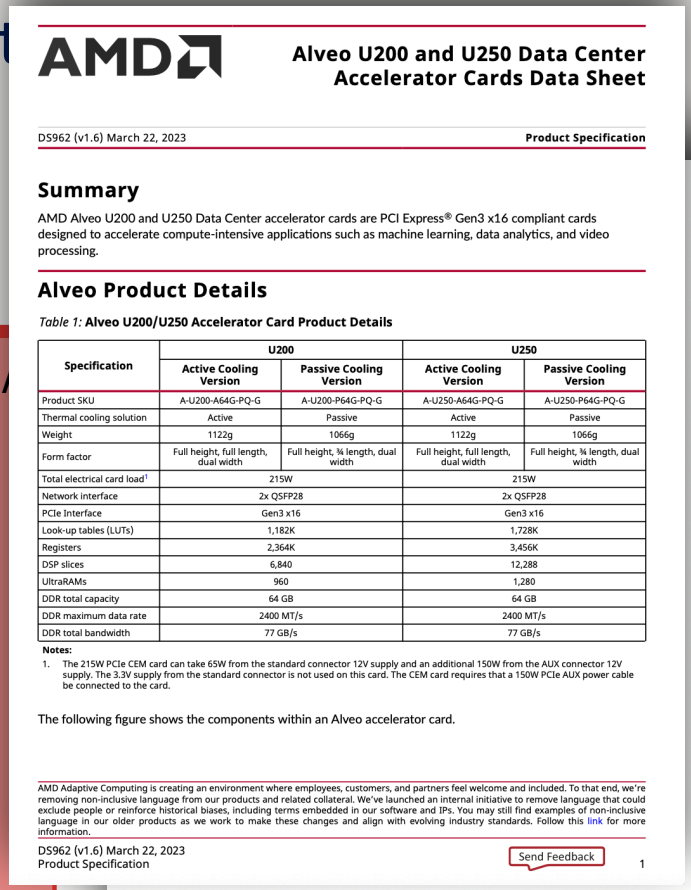




Portability through Separation of Concerns

- **Device engineers** develop re-usable platform integrat





AMD

Alveo U200 and U250 Data Center Accelerator Cards Data Sheet

DS962 (v1.6) March 22, 2023 **Product Specification**

Summary

AMD Alveo U200 and U250 Data Center accelerator cards are PCI Express® Gen3 x16 compliant cards designed to accelerate compute-intensive applications such as machine learning, data analytics, and video processing.

Alveo Product Details

Table 1: **Alveo U200/U250 Accelerator Card Product Details**

Specification	U200		U250	
	Active Cooling Version	Passive Cooling Version	Active Cooling Version	Passive Cooling Version
Product SKU	A-U200-A64G-PQ-G	A-U200-P64G-PQ-G	A-U250-A64G-PQ-G	A-U250-P64G-PQ-G
Thermal cooling solution	Active	Passive	Active	Passive
Weight	1122g	1066g	1122g	1066g
Form factor	Full height, full length, dual width	Full height, ¾ length, dual width	Full height, full length, dual width	Full height, ¾ length, dual width
Total electrical card load[1]	215W		215W	
Network interface	2x QSFP28		2x QSFP28	
PCIe Interface	Gen3 x16		Gen3 x16	
Look-up tables (LUTs)	1,182K		1,728K	
Registers	2,364K		3,456K	
DSP slices	6,840		12,288	
UltraRAMs	960		1,280	
DDR total capacity	64 GB		64 GB	
DDR maximum data rate	2400 MT/s		2400 MT/s	
DDR total bandwidth	77 GB/s		77 GB/s	

Notes:
1. The 215W PCIe CEM card can take 65W from the standard connector 12V supply and an additional 150W from the AUX connector 12V supply. The 3.3V supply from the standard connector is not used on this card. The CEM card requires that a 150W PCIe AUX power cable be connected to the card.

The following figure shows the components within an Alveo accelerator card.

AMD Adaptive Computing is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

DS962 (v1.6) March 22, 2023
Product Specification

Send Feedback

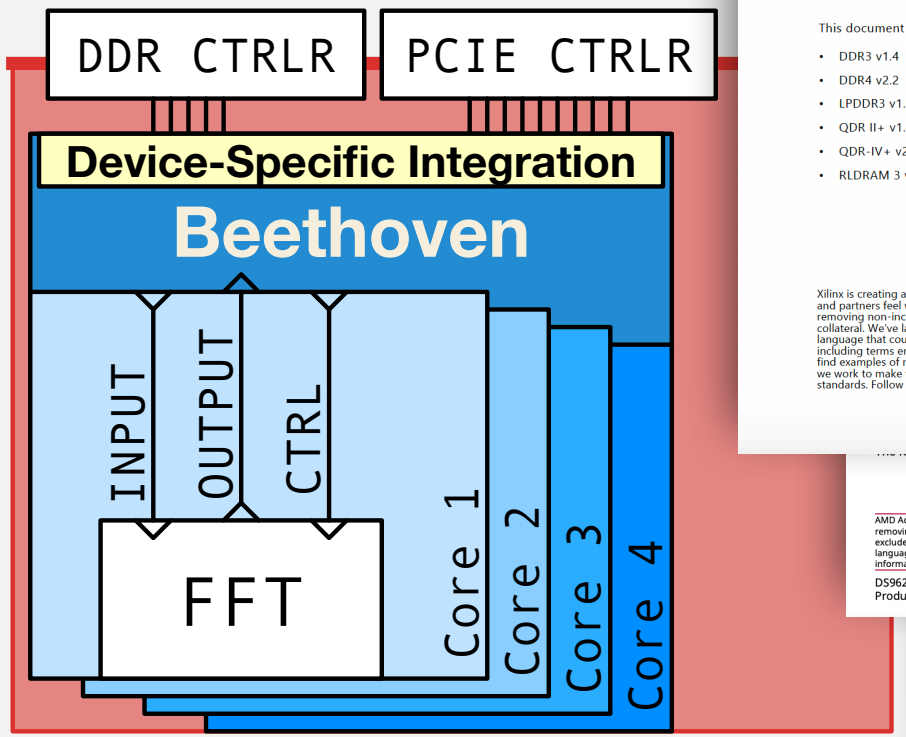
1

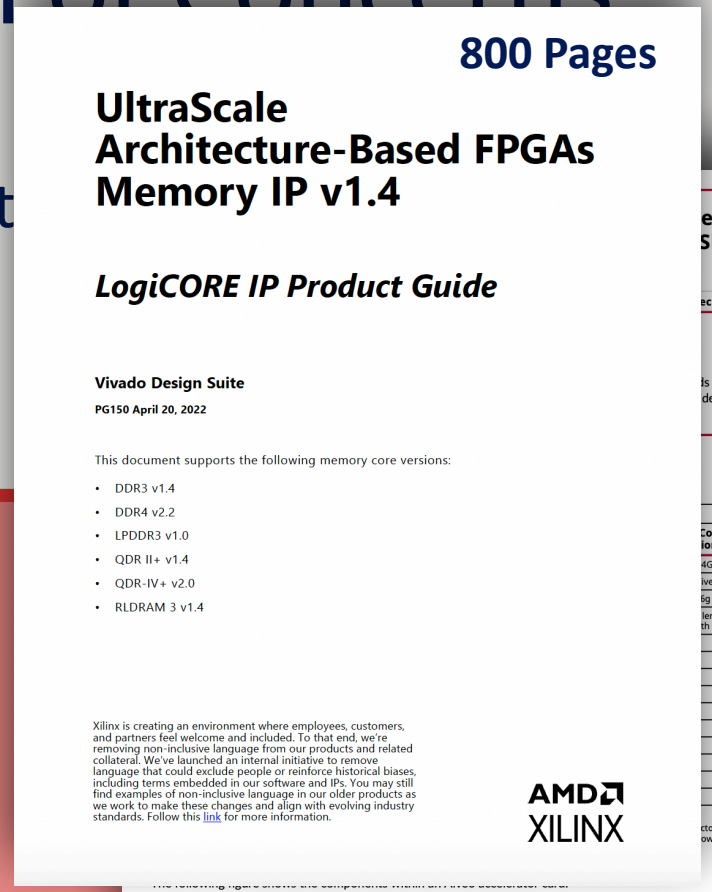
Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

AMD XILINX

Portability through Separation of Concerns

- **Device engineers** develop re-usable platforms int




800 Pages

UltraScale Architecture-Based FPGAs Memory IP v1.4

LogiCORE IP Product Guide

Vivado Design Suite

PG150 April 20, 2022

This document supports the following memory core versions:

- DDR3 v1.4
- DDR4 v2.2
- LPDDR3 v1.0
- QDR II+ v1.4
- QDR-IV+ v2.0
- RLDRAM 3 v1.4

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

AMD XILINX

AMD Adaptive Computing is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

DS962 (v1.6) March 22, 2023
Product Specification

Send Feedback

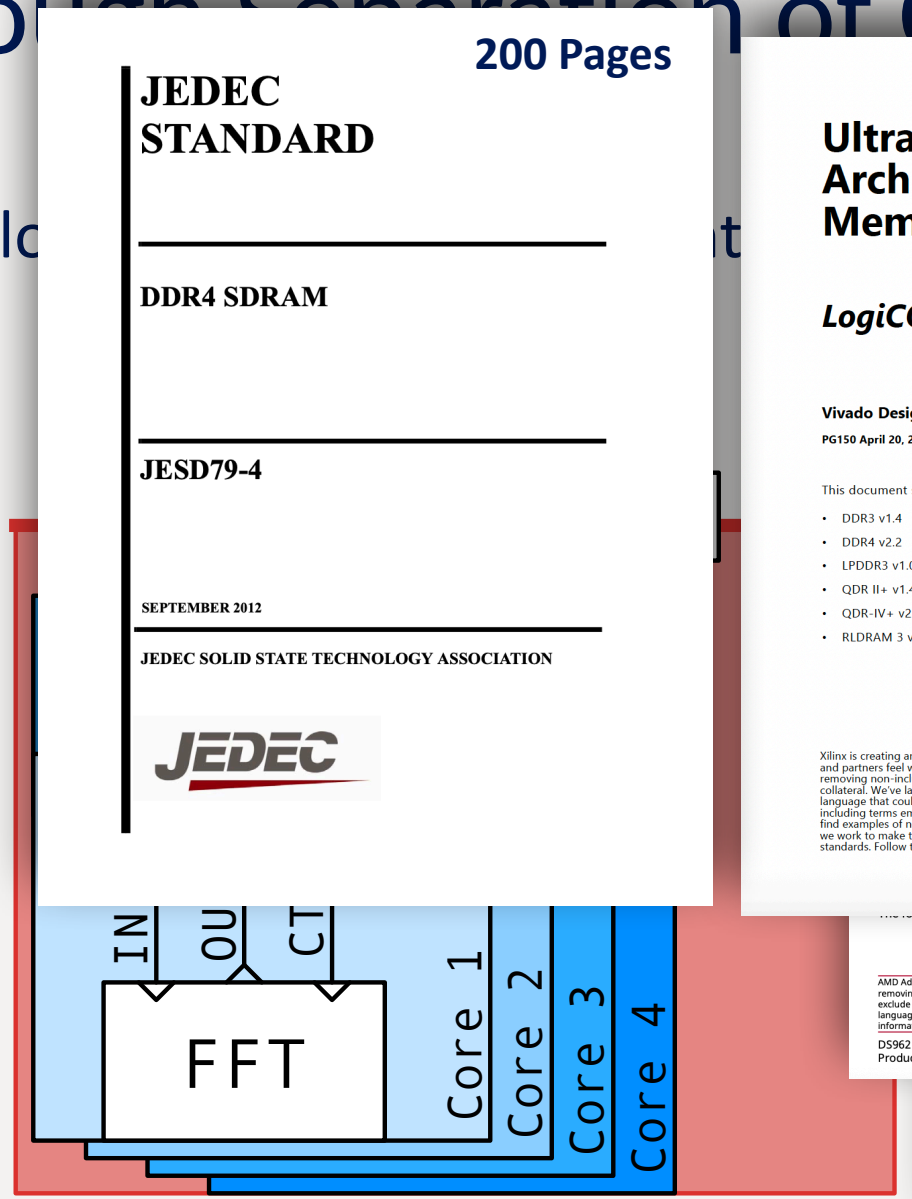
1

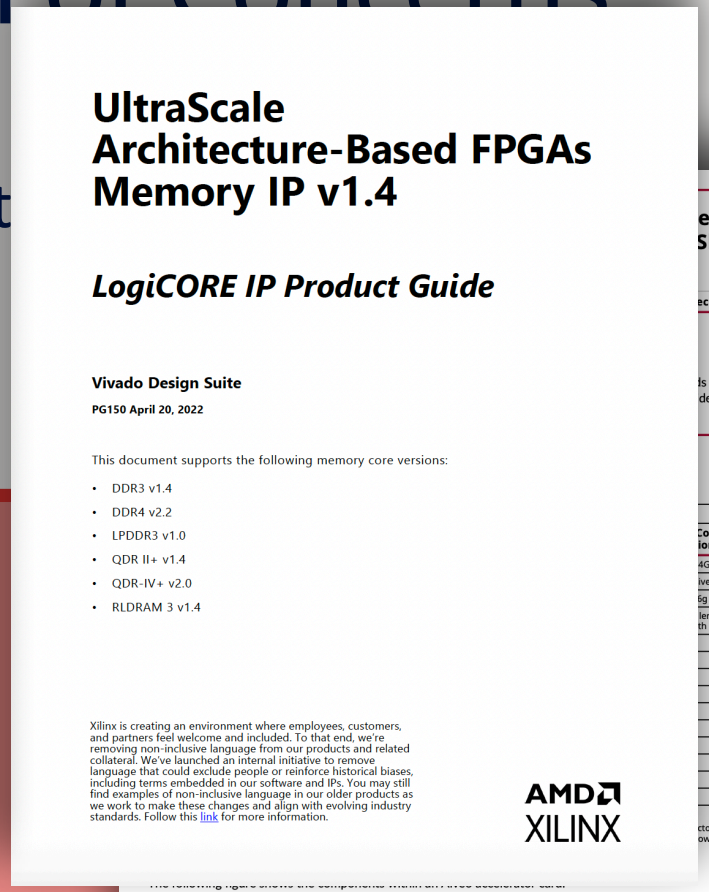
Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

AMD XILINX

Portability through Separation of Concerns

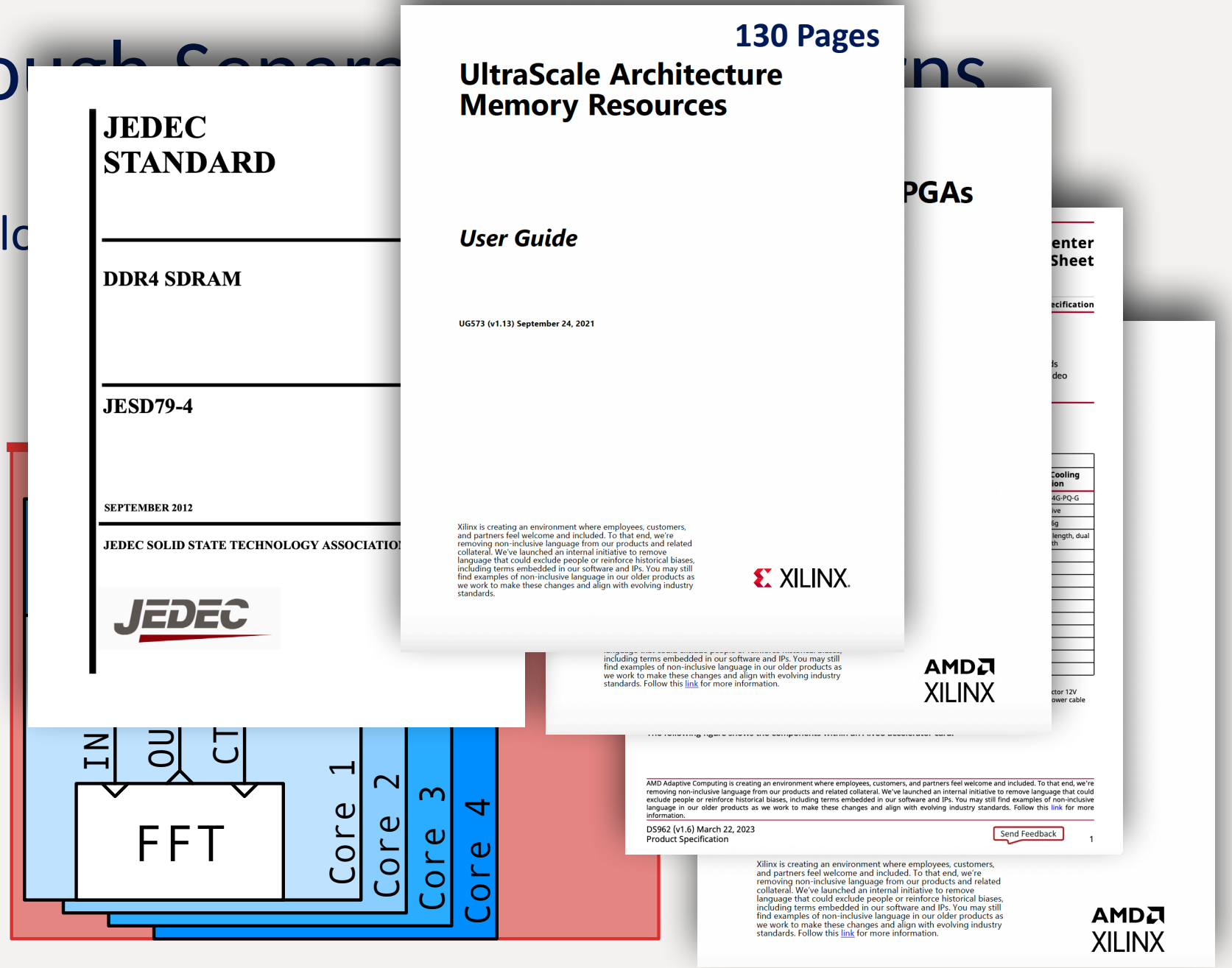
- **Device engineers** develo





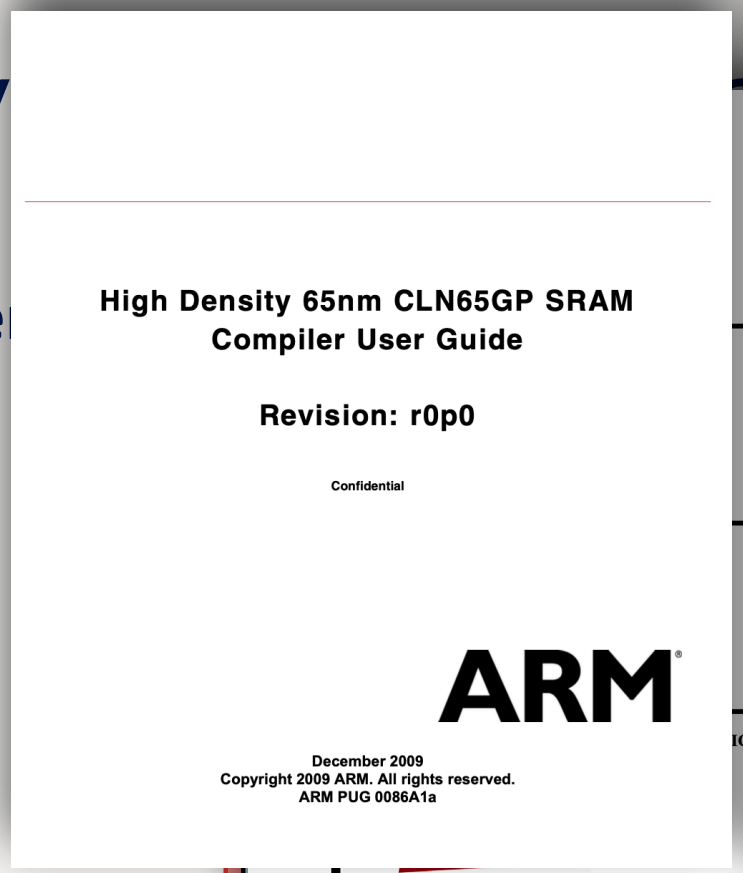
Portability through Separa ns

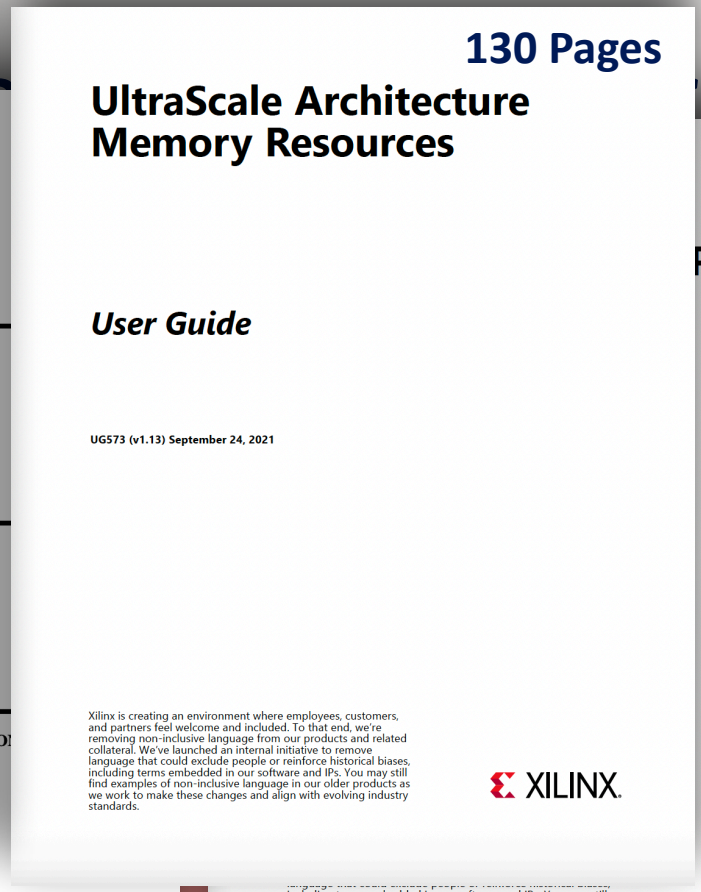
- **Device engineers** develo

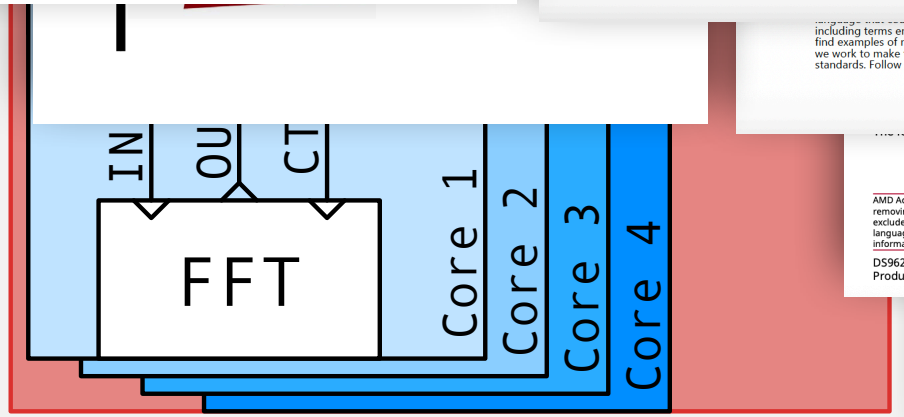


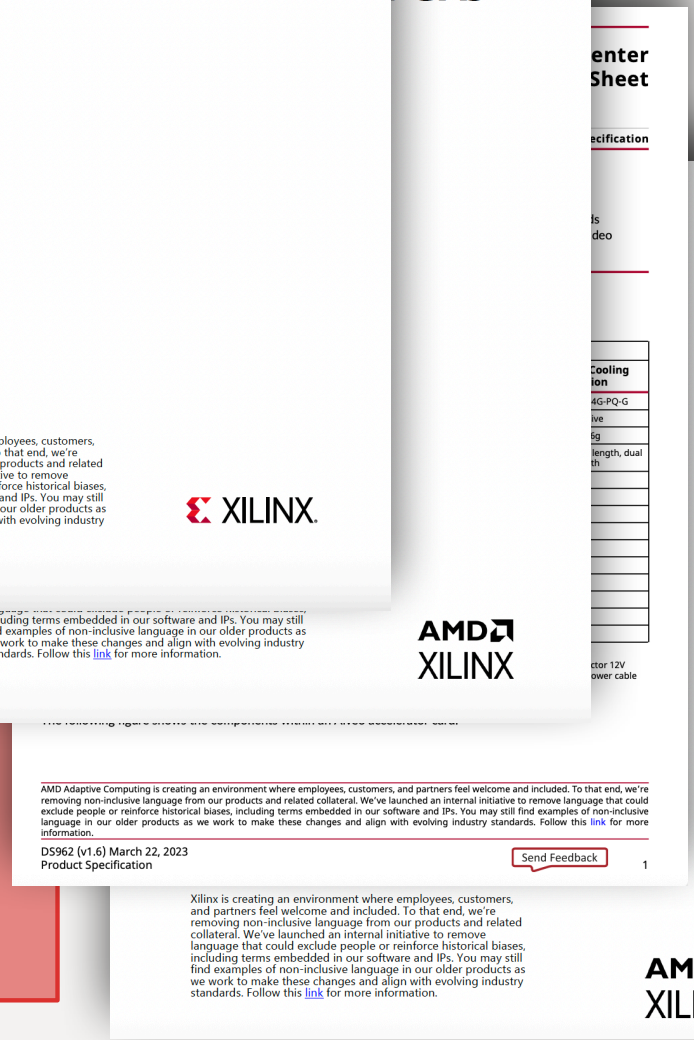
Portability

- **Device enginee**



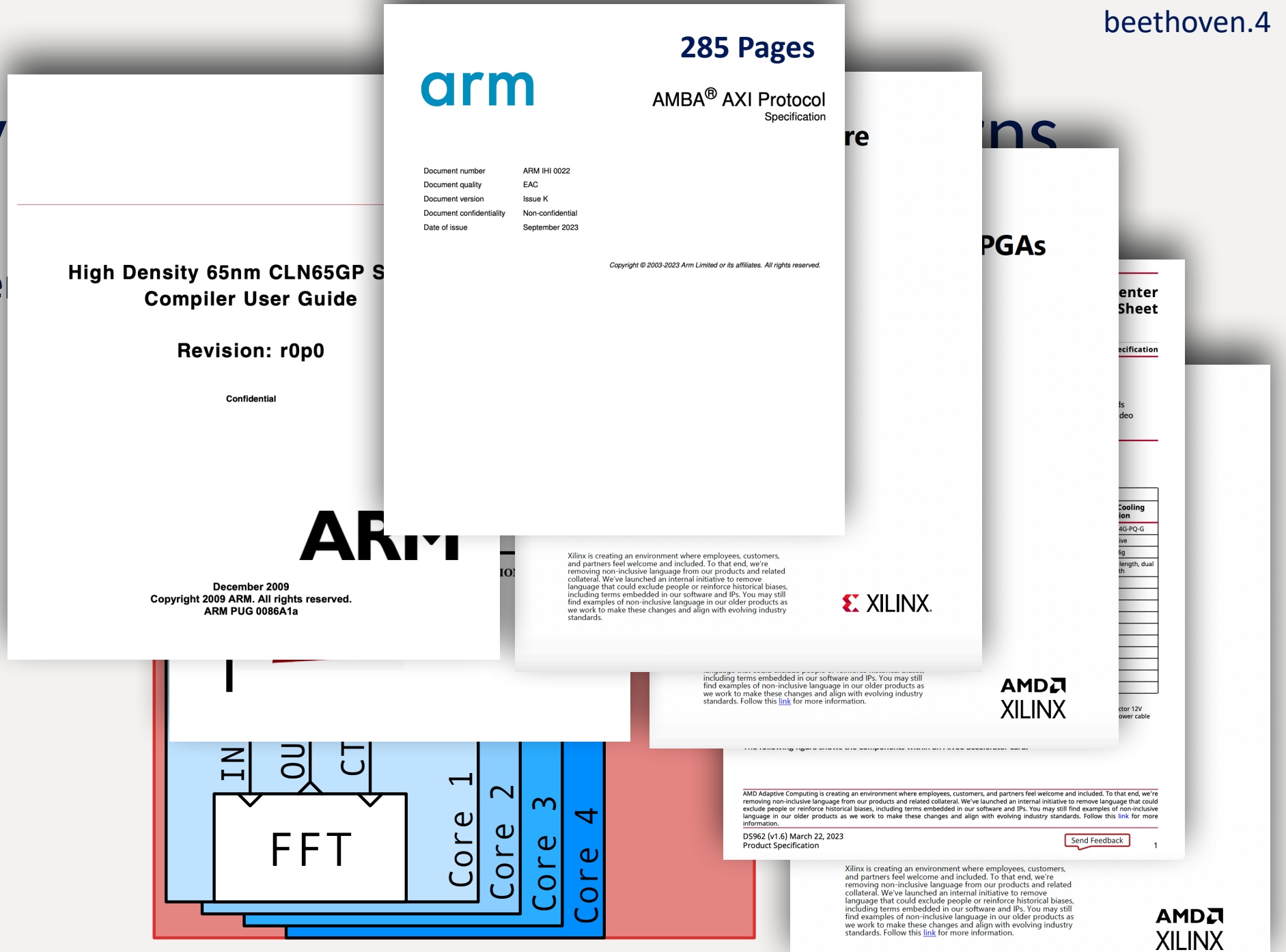






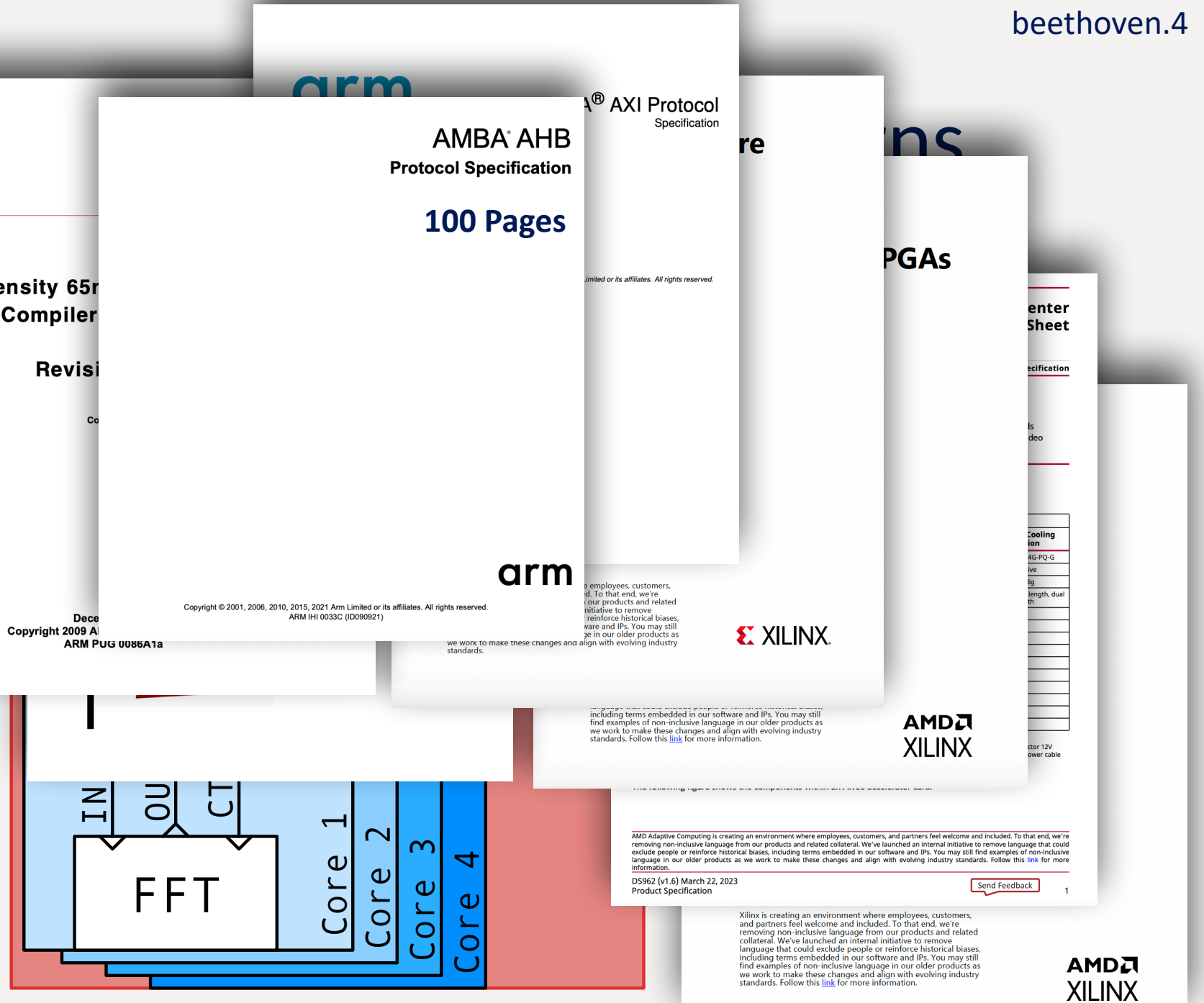
Portability

- **Device enginee**



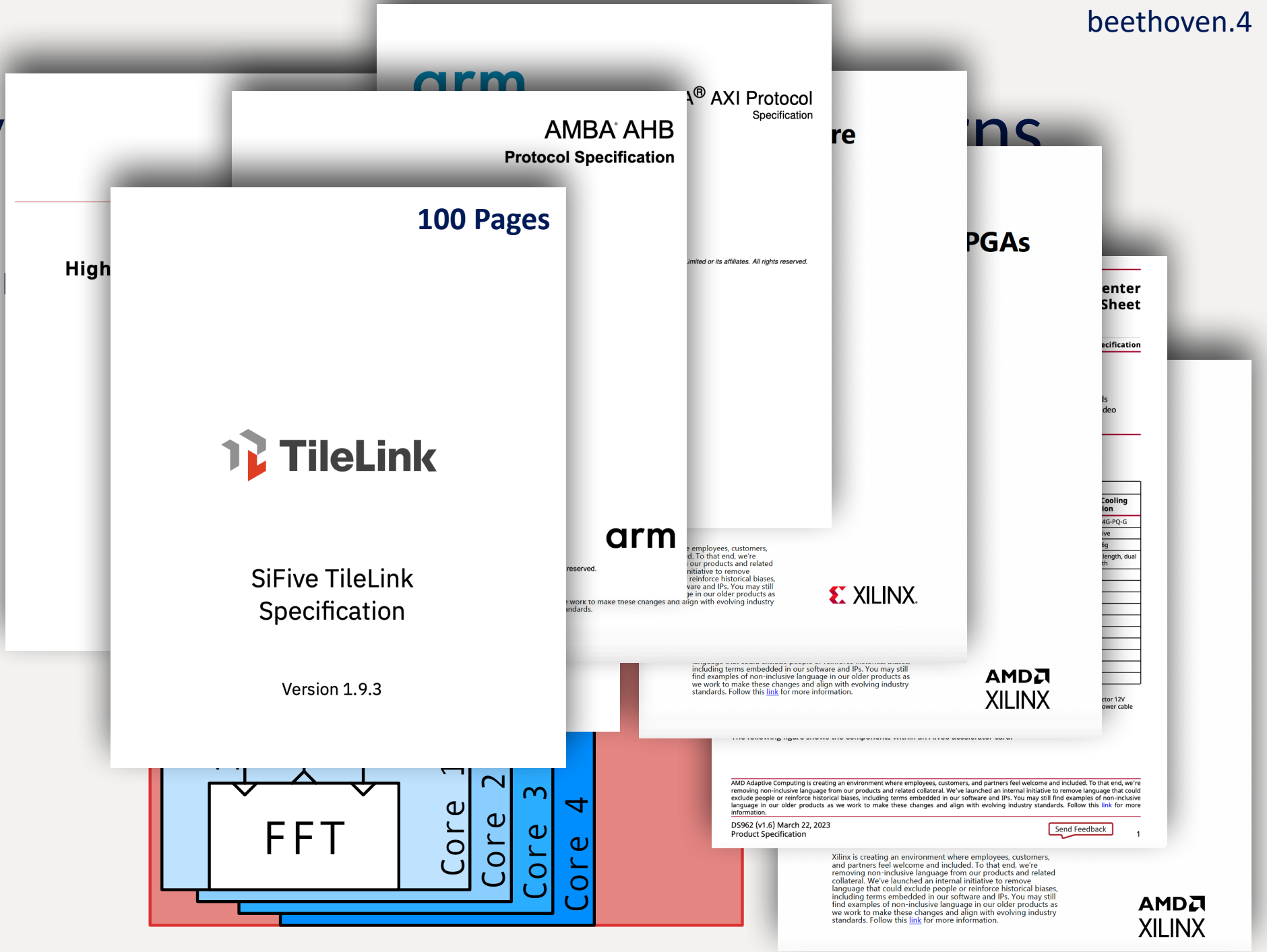
Portability

- **Device enginee**



Portability

- **Device enginee**



Portability

- Device enginee

IEEE Standards

IEEE Std 1364-2001
(Revision of
IEEE Std 1364-1995)

IEEE Standard Verilog® Hardware Description Language

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE

Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

28 September 2001

Print: SH94921
PDF: SS94921

800 Pages

The Verilog PLI Handbook

A User's Guide
and
Comprehensive Reference
on the
Verilog Programming Language Interface

AMBA® AHB
Protocol Specification

arm

XILINX

AMD
XILINX

FFT

Core 4

Portability

- Device enginee

Beethoven

System, Code Structures, and Abstractions

Chris Kjellqvist

ISCA 2026 Tutorial

Duke

Chisel HDL

Beethoven uses Chisel, a high-level RTL hardware design language

- Built on Scala, providing functional programming patterns and structures
- Compiles to Verilog

```
wire [15:0] data_i [0:127];
reg [15:0] data_o [0:127];
integer i;
always @(posedge clk) begin
  for (i=0; i<128; i=i+1) begin
    data_o = data_i - 2;
  end
end
```

Verilog implementation of map pattern

```
val data_i = Wire(Vec(128, UInt(16.W)))
val data_o = data_i.map(_ - 2.U)
```

Chisel is built on Scala and incorpates the functional language patterns in hardware development

Duke

Chisel HDL

Beethoven uses Chisel, a high-level RTL hardware design language

- Built on Scala, providing functional programming patterns and structures
- Compiles to Verilog
- *Beethoven supports Verilog!*

```
wire [15:0] data_i [0:127];
reg [15:0] data_o [0:127];
integer i;
always @(posedge clk) begin
  for (i=0; i<128; i=i+1) begin
    data_o = data_i - 2;
  end
end
```

Verilog implementation of map pattern

```
val data_i = Wire(Vec(128, UInt(16.W)))
val data_o = data_i.map(_ - 2.U)
```

Chisel is built on Scala and incorpates the functional language patterns in hardware development

Duke

Chisel HDL

Beethoven uses Chisel, a high-level RTL hardware design language

- Built on Scala, providing functional programming patterns and structures
- Compiles to Verilog
- *Beethoven supports Verilog!*

Why Chisel? Verilog has limitations

- `` `generate` `` can't do it all

```
wire [15:0] data_i [0:127];
reg [15:0] data_o [0:127];
integer i;
always @(posedge clk) begin
  for (i=0; i<128; i=i+1) begin
    data_o = data_i - 2;
  end
end
```

Verilog implementation of map pattern

```
val data_i = Wire(Vec(128, UInt(16.W)))
val data_o = data_i.map(_ - 2.U)
```

Chisel is built on Scala and incorpates the functional language patterns in hardware development

Duke

Chisel HDL

Beethoven uses Chisel, a high-level RTL hardware design language

- Built on Scala, providing functional programming patterns and structures
- Compiles to Verilog
- *Beethoven supports Verilog!*

Why Chisel? Verilog has limitations

- `generate` can't do it all
- SoC floorplanning, backend file generation
- Verilog annotations
- Automatic generation of external IPs

```
wire [15:0] data_i [0:127];
reg [15:0] data_o [0:127];
integer i;
always @(posedge clk) begin
  for (i=0; i<128; i=i+1) begin
    data_o = data_i - 2;
  end
end
```

Verilog implementation of map pattern

```
val data_i = Wire(Vec(128, UInt(16.W)))
val data_o = data_i.map(_ - 2.U)
```

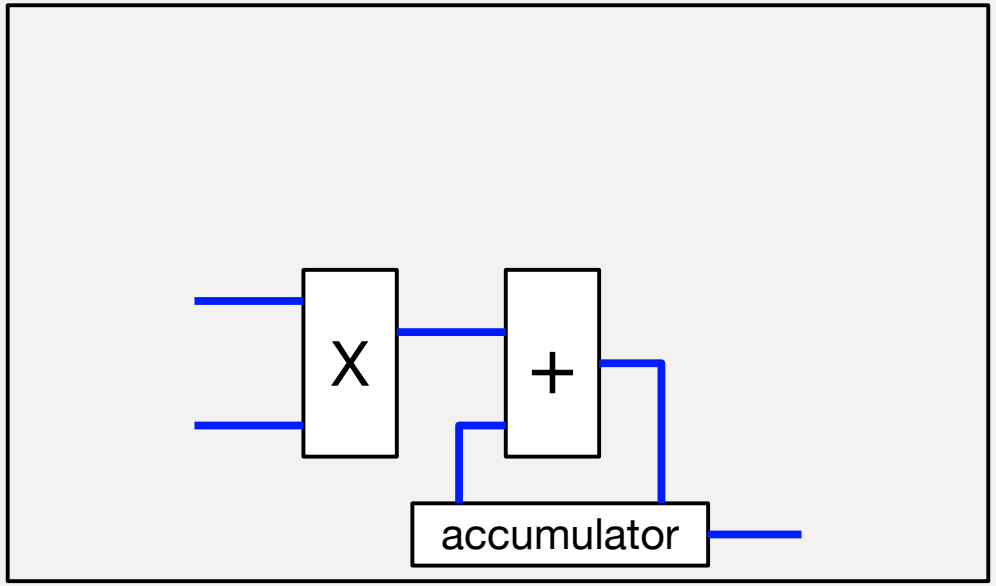
Chisel is built on Scala and incorpates the functional language patterns in hardware development

Duke

Example Introduction

Vector Dot Product Accelerator

$$o = \sum_{i}^{l} A_i \cdot B_i$$



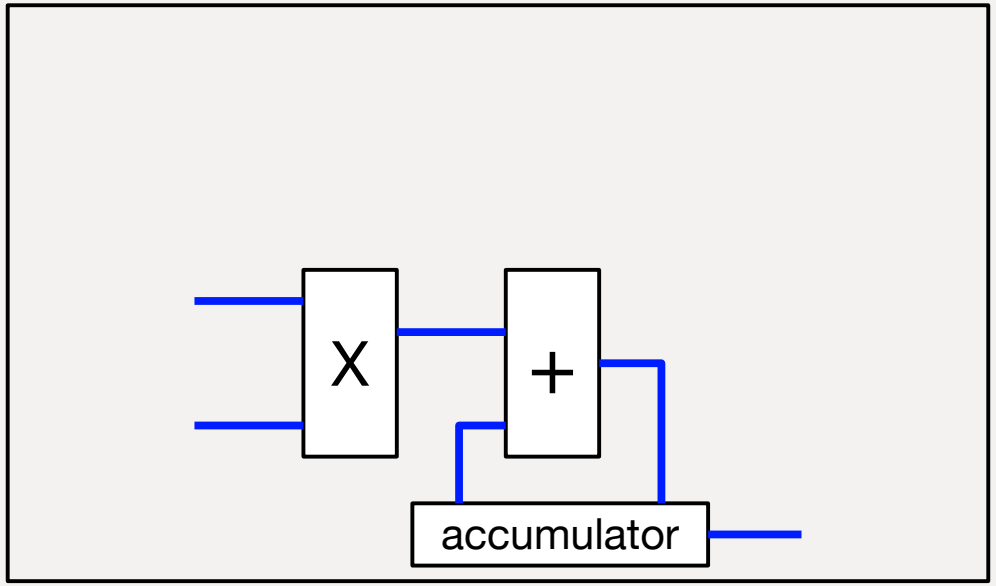
Duke

Vector Dot Product Accelerator

$$o = \sum_{i}^{l} A_i \cdot B_i$$

What memory access is needed?

- Data Inputs: Read Streams
- Data Outputs: Single datum



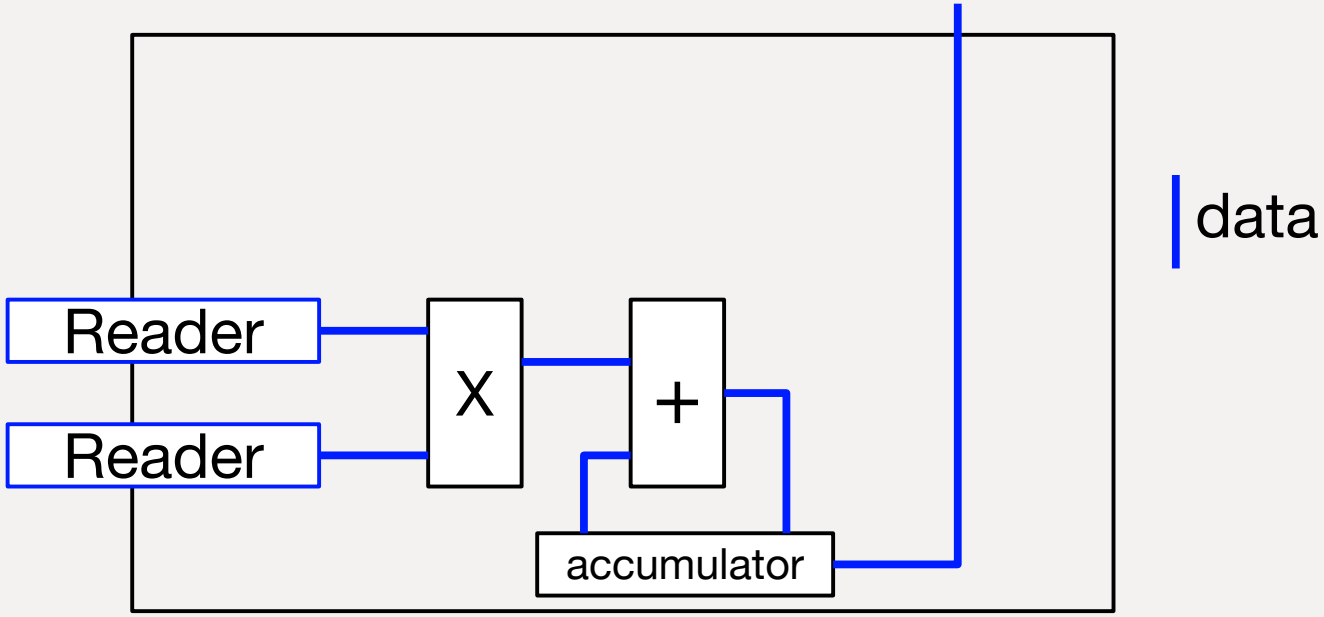
Duke

Vector Dot Product Accelerator

$$o = \sum_{i}^{l} A_i \cdot B_i$$

What memory access is needed?

- Data Inputs: Read Streams
- Data Outputs: Single datum



Duke

Vector Dot Product Accelerator

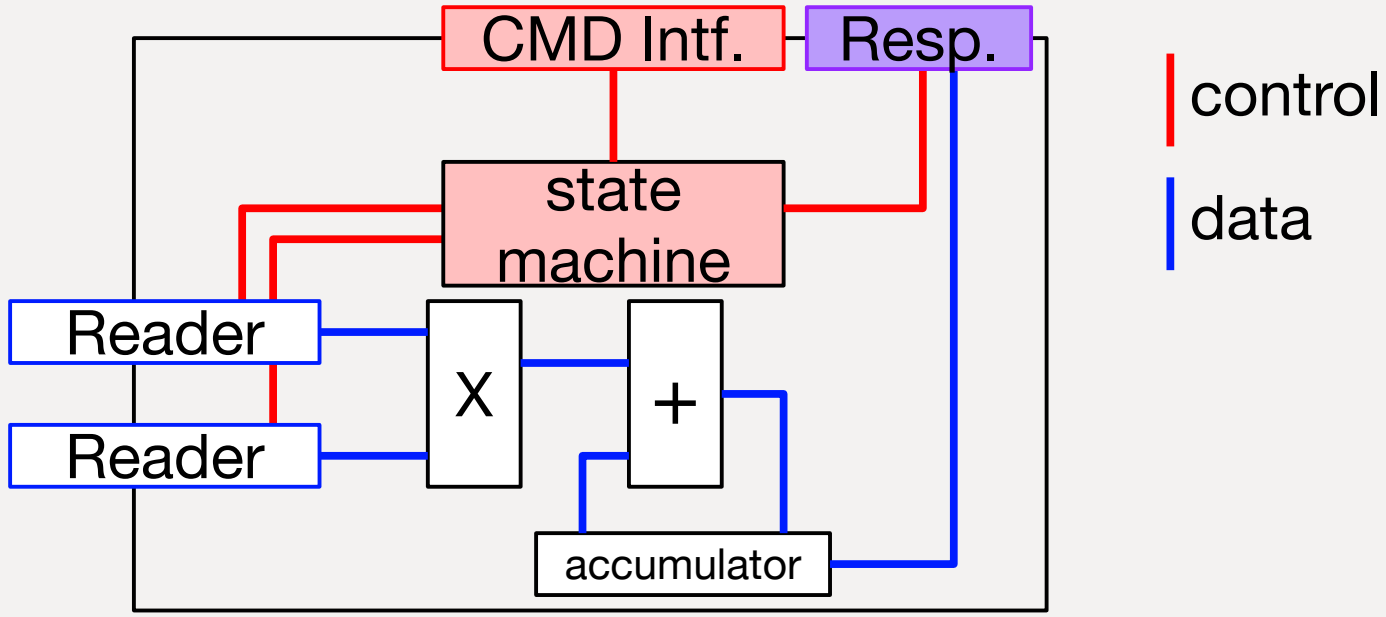
$$o = \sum_{i}^{l} A_i \cdot B_i$$

What memory access is needed?

- Data Inputs: Read Streams
- Data Outputs: Single datum

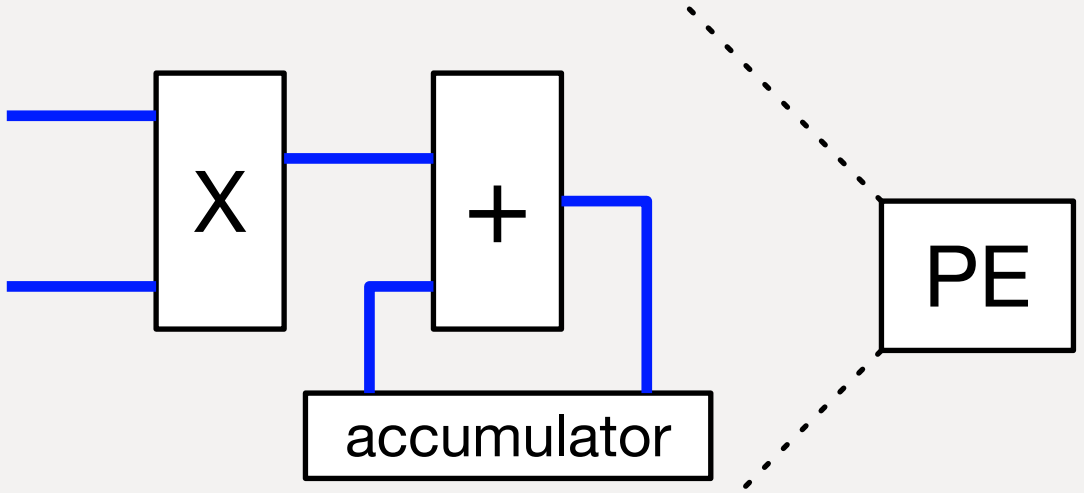
How do we use it? Control?

- Host Interfaces

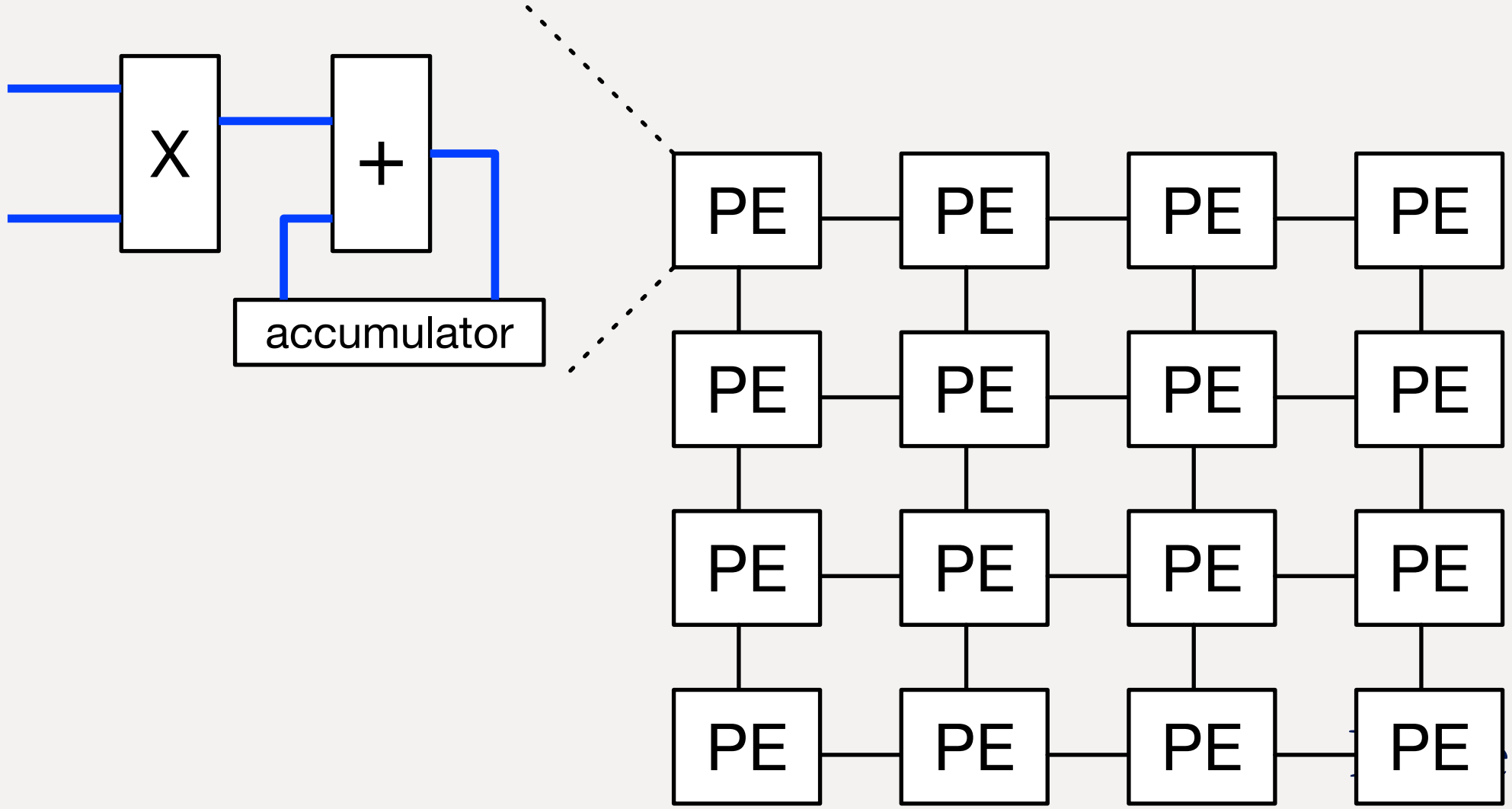


Duke

Systolic Array

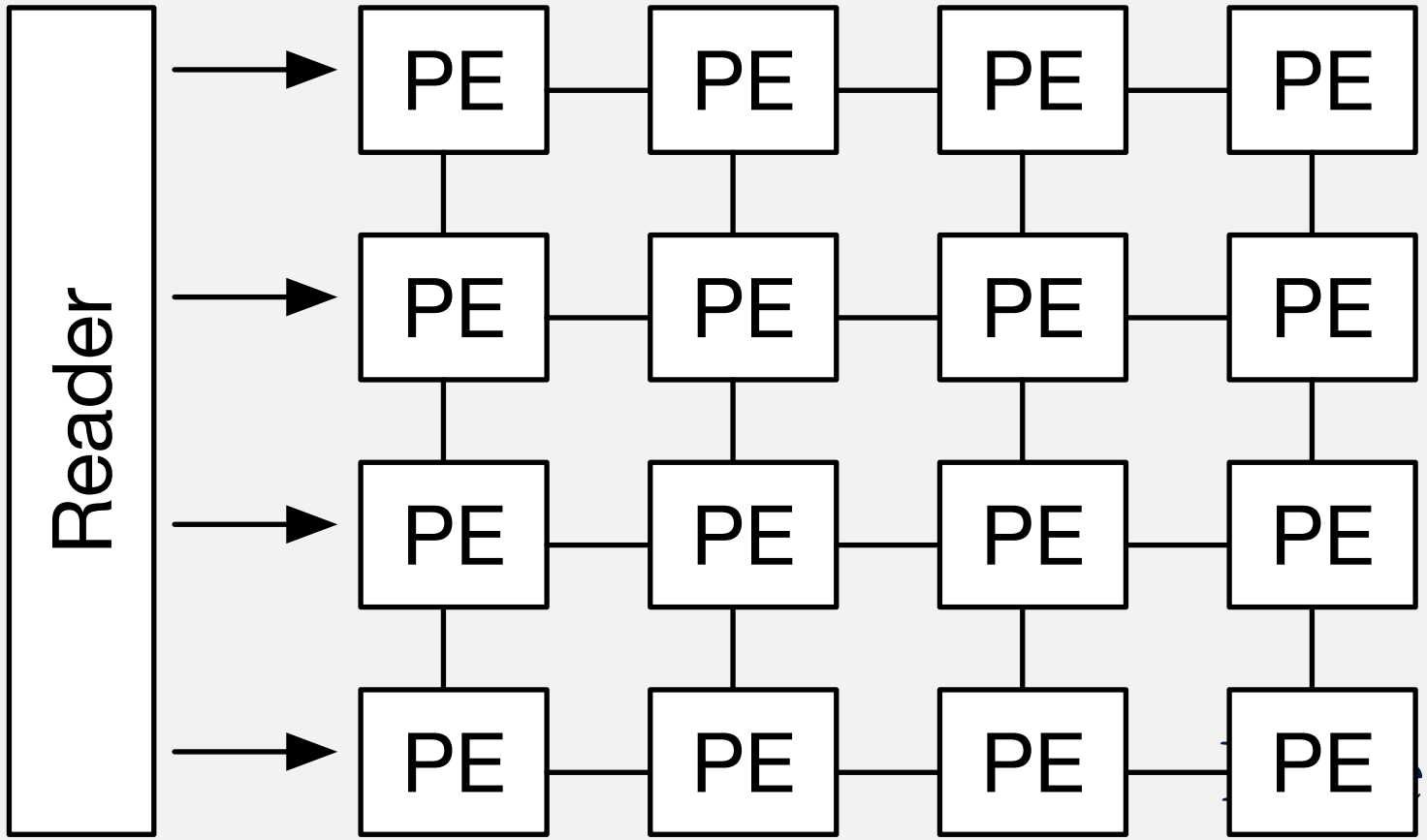


Systolic Array



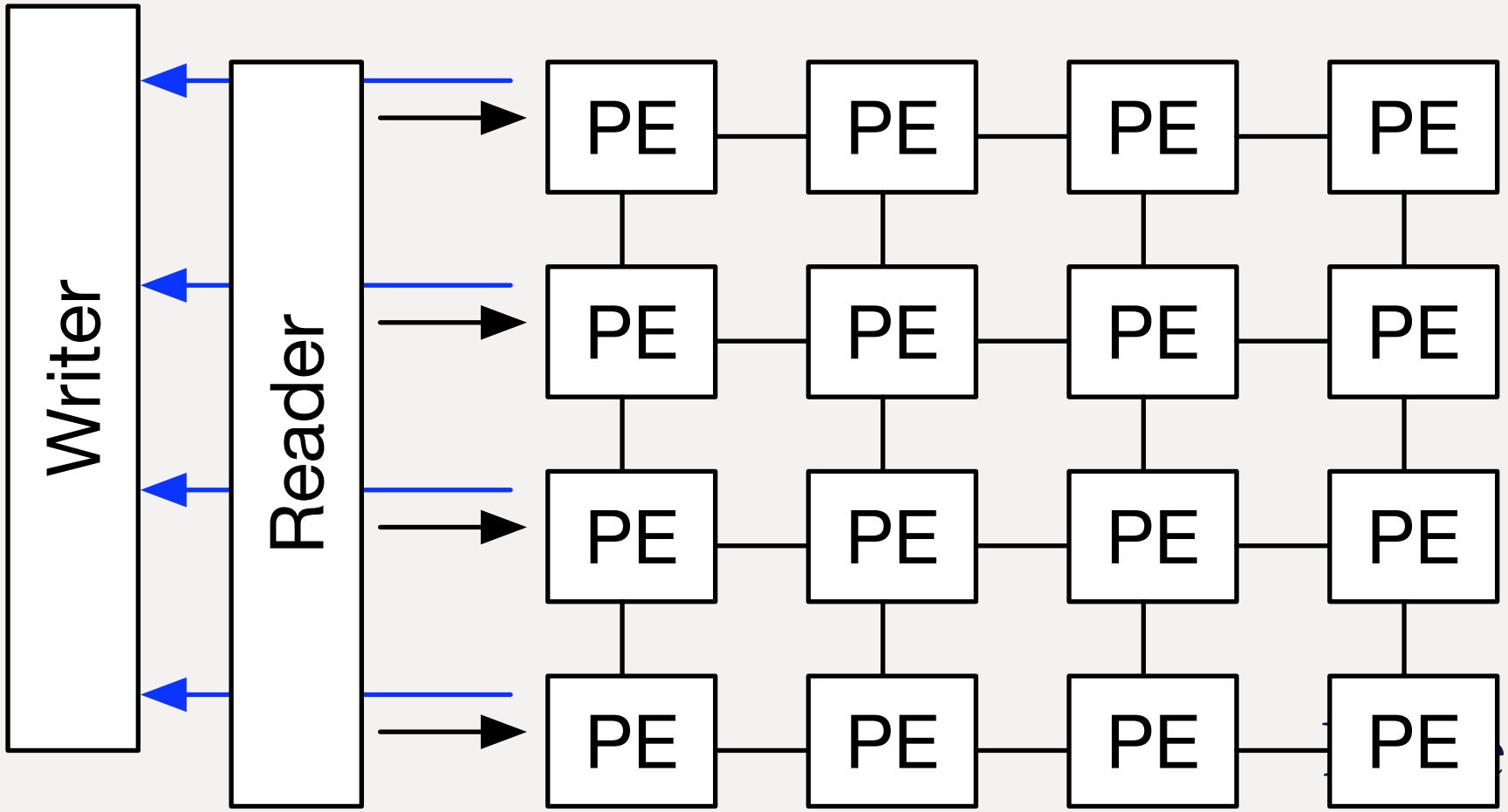
Systolic Array

- Reader streams activations from memory



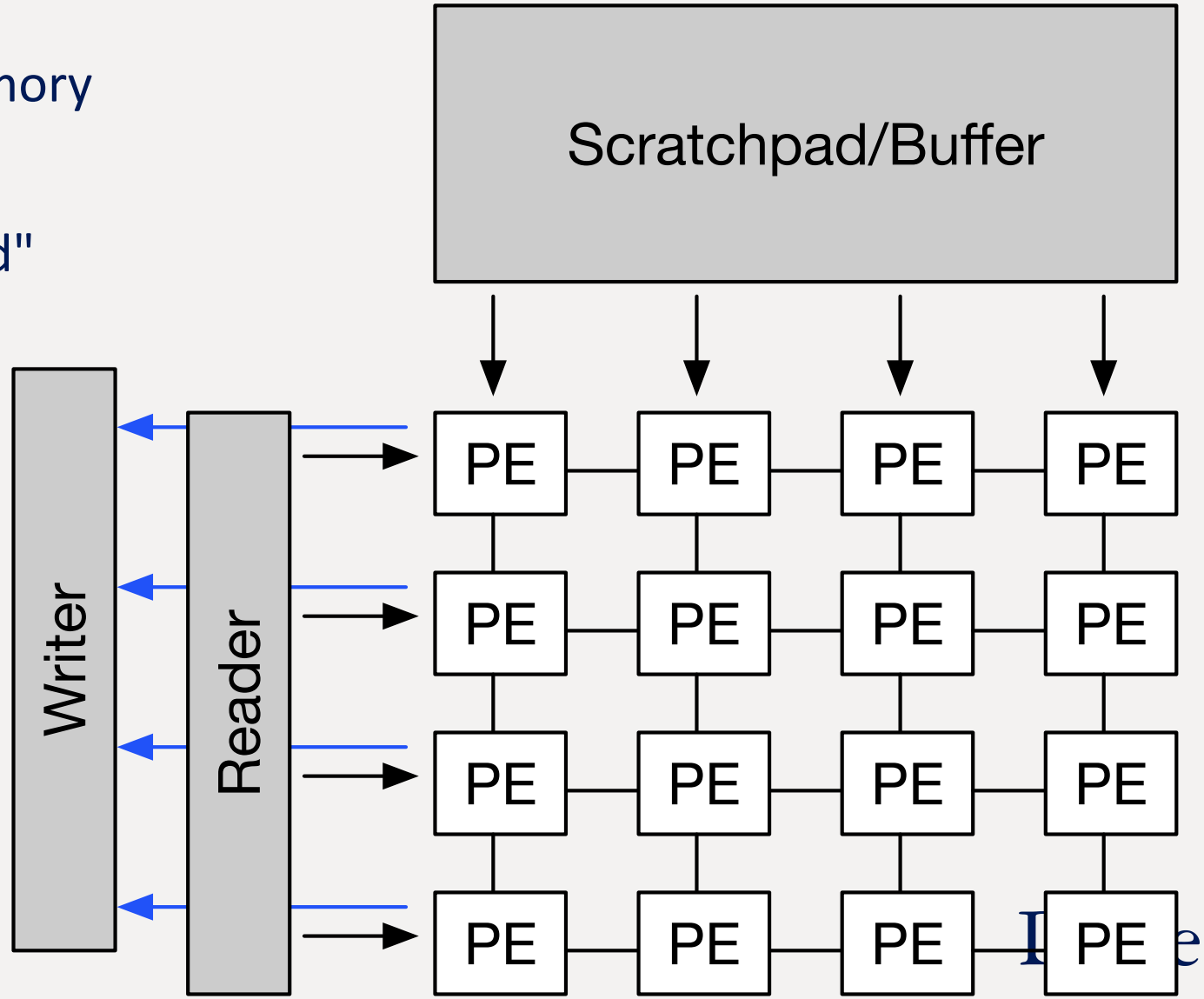
Systolic Array

- Reader streams activations from memory
- Writer writes results



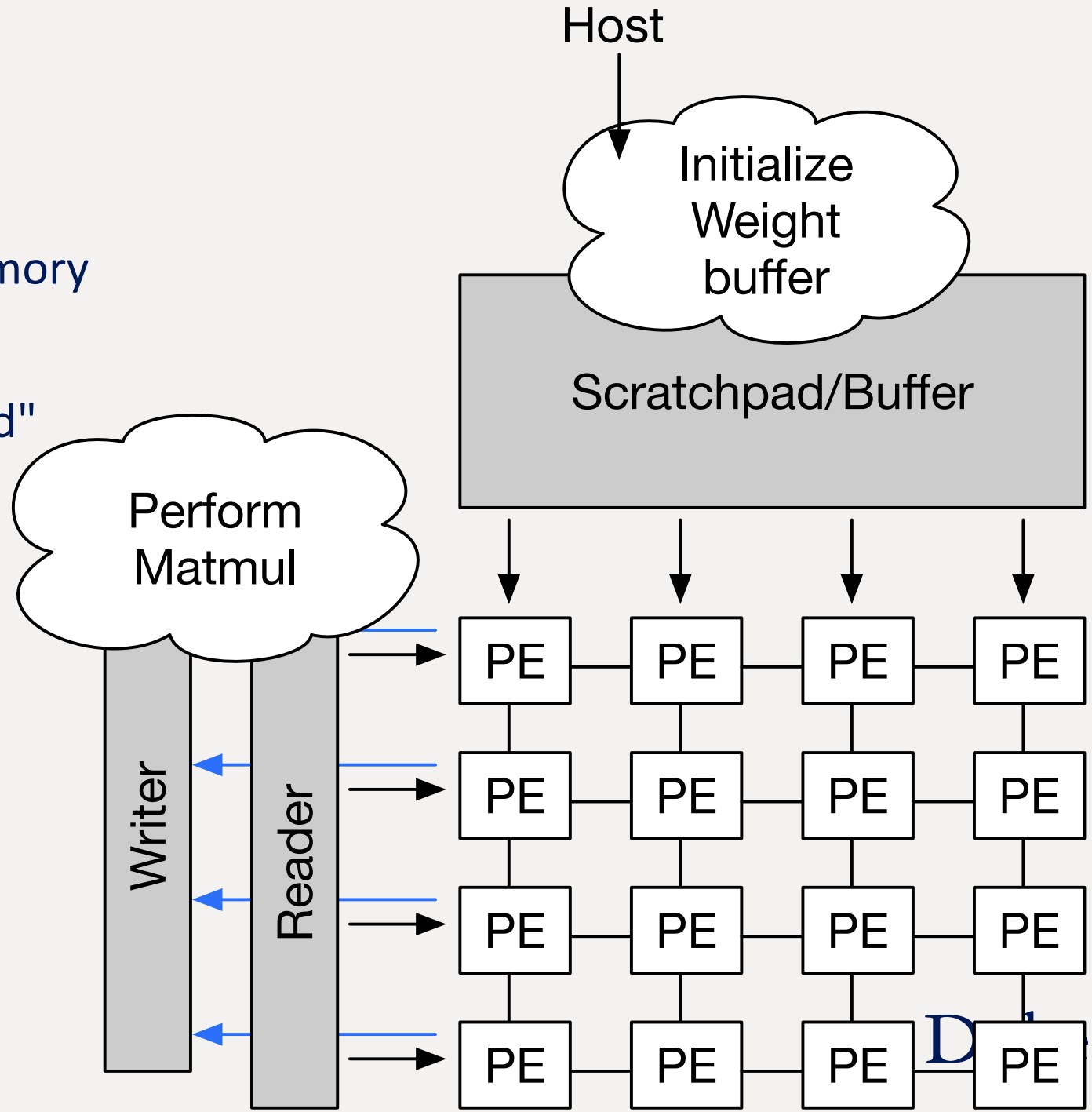
Systolic Array

- Reader streams activations from memory
- Writer writes results
- Weights are buffered in a "Scratchpad"



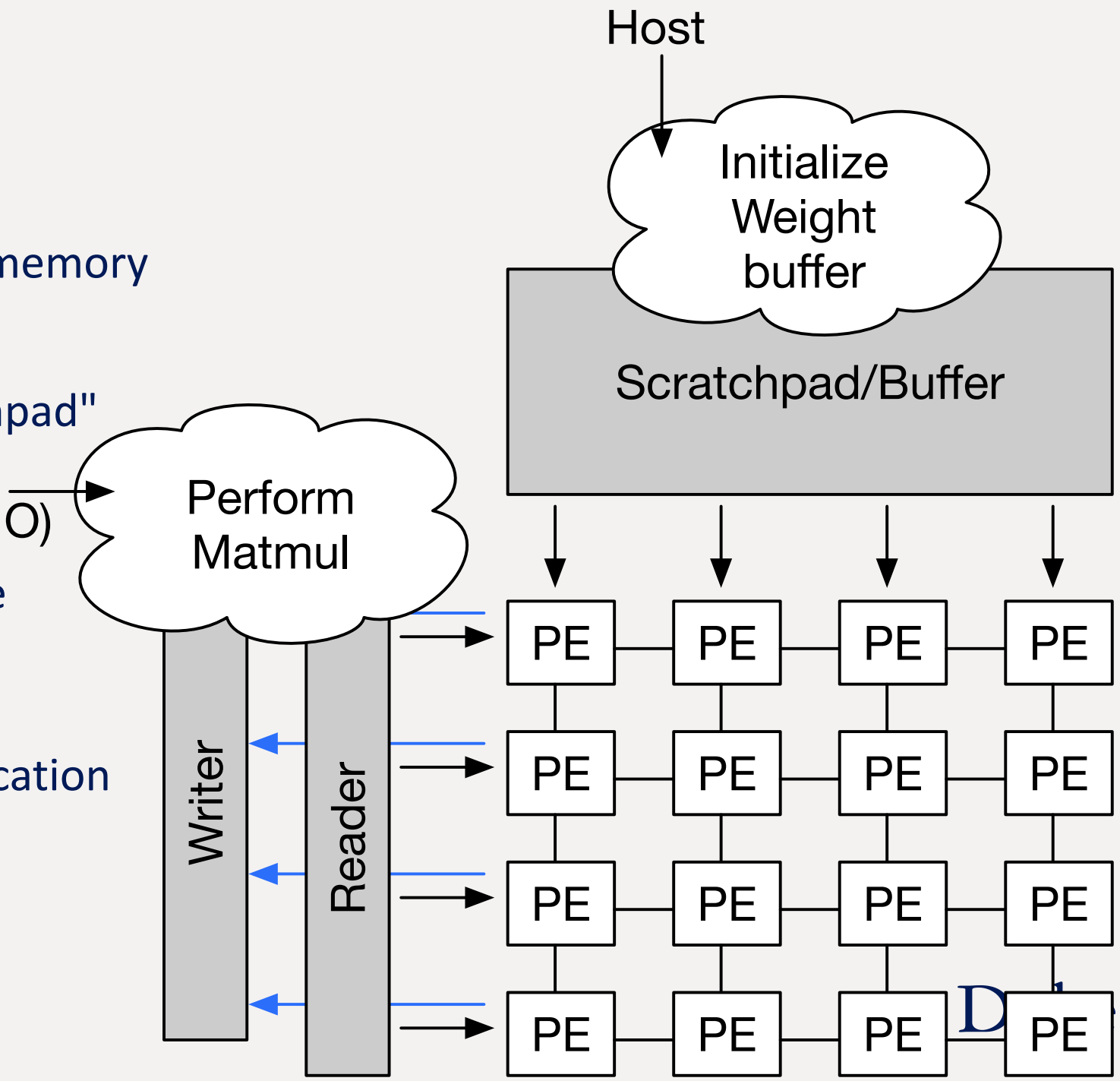
Systolic Array

- Reader streams activations from memory
- Writer writes results
- Weights are buffered in a "Scratchpad"
- Weight buffer has a state machine for initializing its contents



Systolic Array

- Reader streams activations from memory
- Writer writes results
- Weights are buffered in a "Scratchpad"
- Weight buffer has a state machine for initializing its contents
- The matmul path has a separate state machine and host communication path



Accelerator Configuration

Declaring Memory Interfaces, Integrating multiple accelerator cores, boilerplate

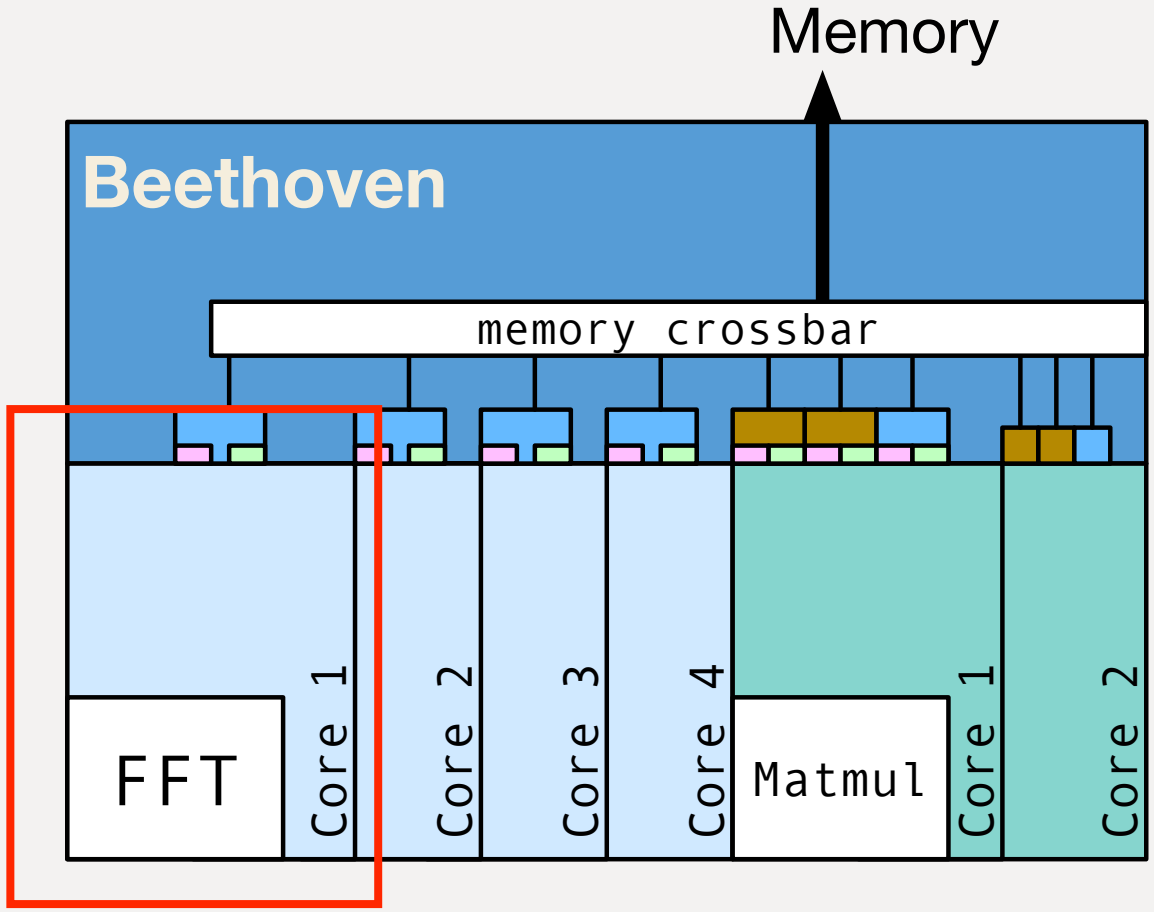
Duke

Accelerator Topology

- How does a real accelerator system look like?

Accelerator Topology

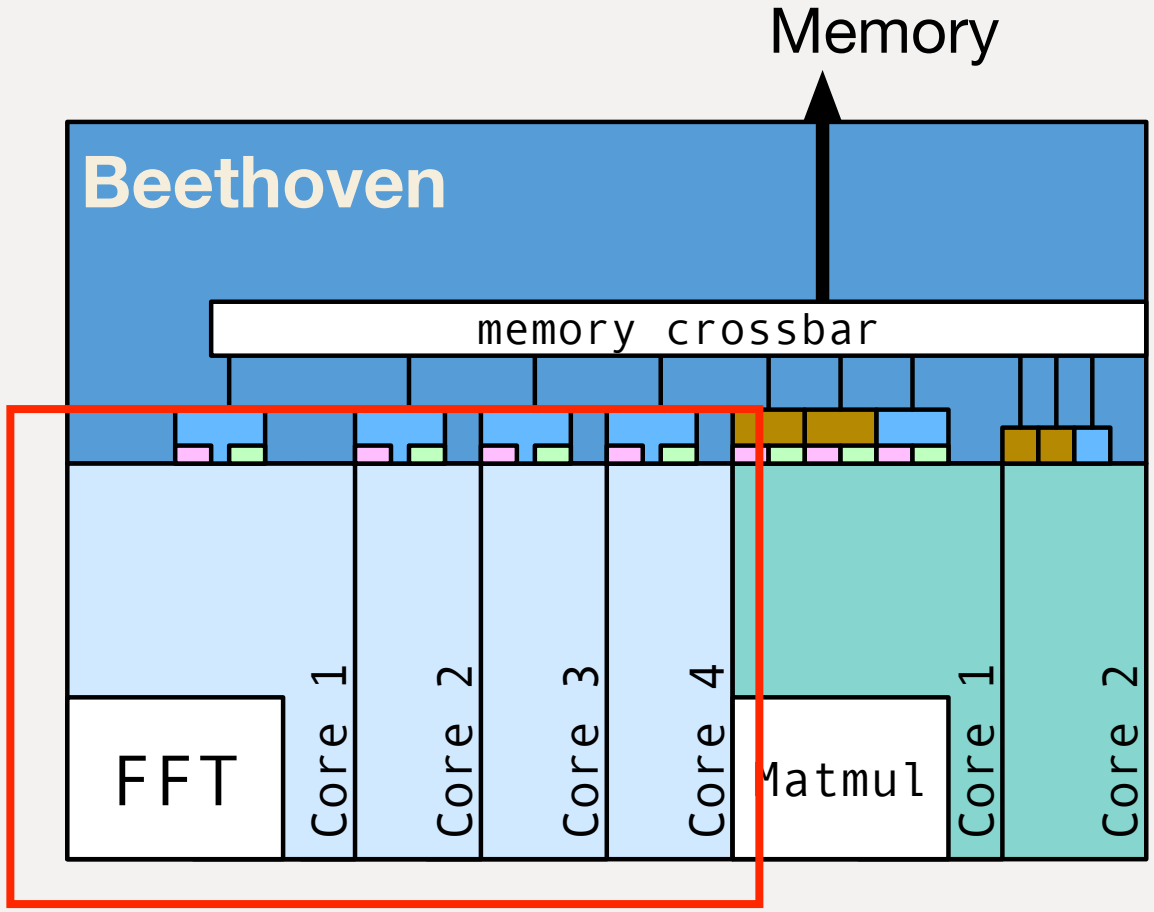
- How does a real accelerator system look like?
  - your accelerator design



Duke

Accelerator Topology

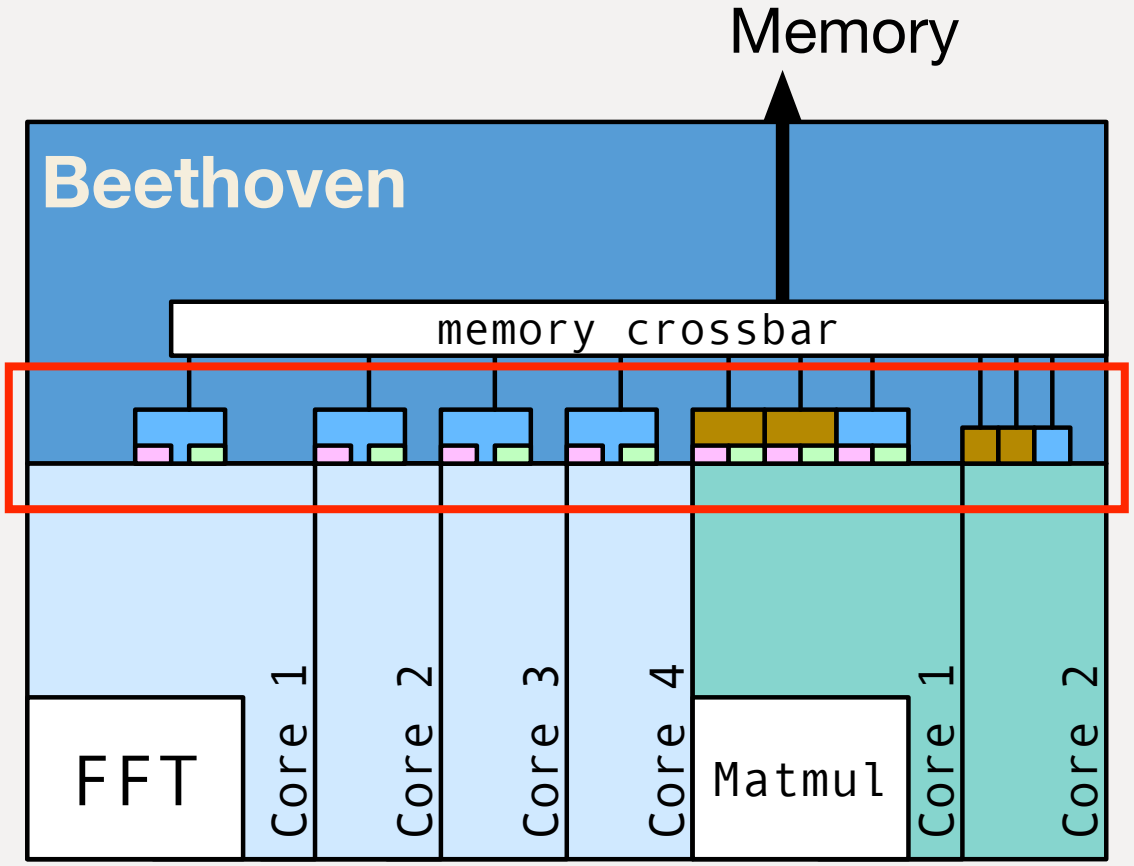
- How does a real accelerator system look like?
  - your accelerator design
  - Multiple accelerator cores?



Duke

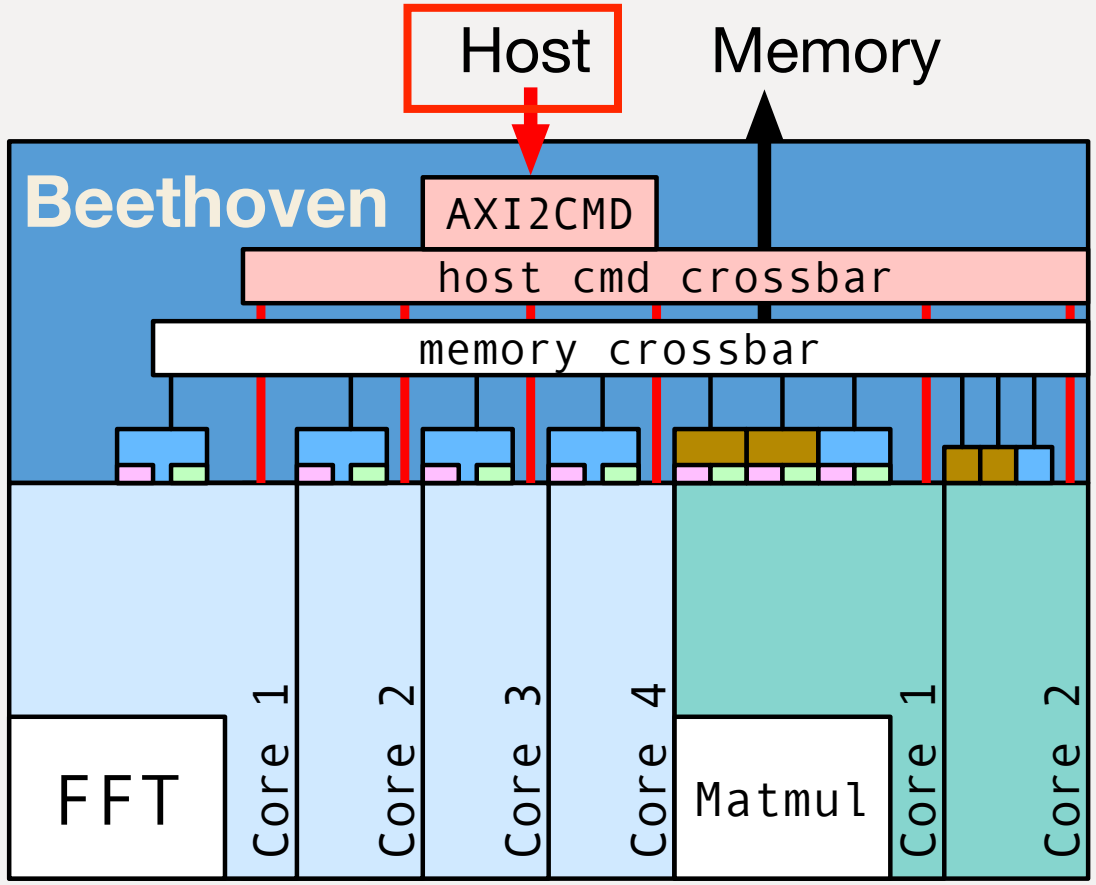
Accelerator Topology

- How does a real accelerator system look like?
  - your accelerator design
  - Multiple accelerator cores?
  - Their memory interfaces: readers/writers



Accelerator Topology

- How does a real accelerator system look like?
  - your accelerator design
  - Multiple accelerator cores?
  - Their memory interfaces: readers/writers
  - Host-Accelerator Interface



Locate your file

```
.
├── Beethoven.toml
├── README.md
├── build.sbt
├── hw
│   └── src # <----- DotProductConfig.scala here
├── project
│   ├── build.properties
│   └── target
├── sw
│   ├── CMakeLists.txt
│   └── dot_product_tb.cc # <----- Software Testbench
.....
```

Duke

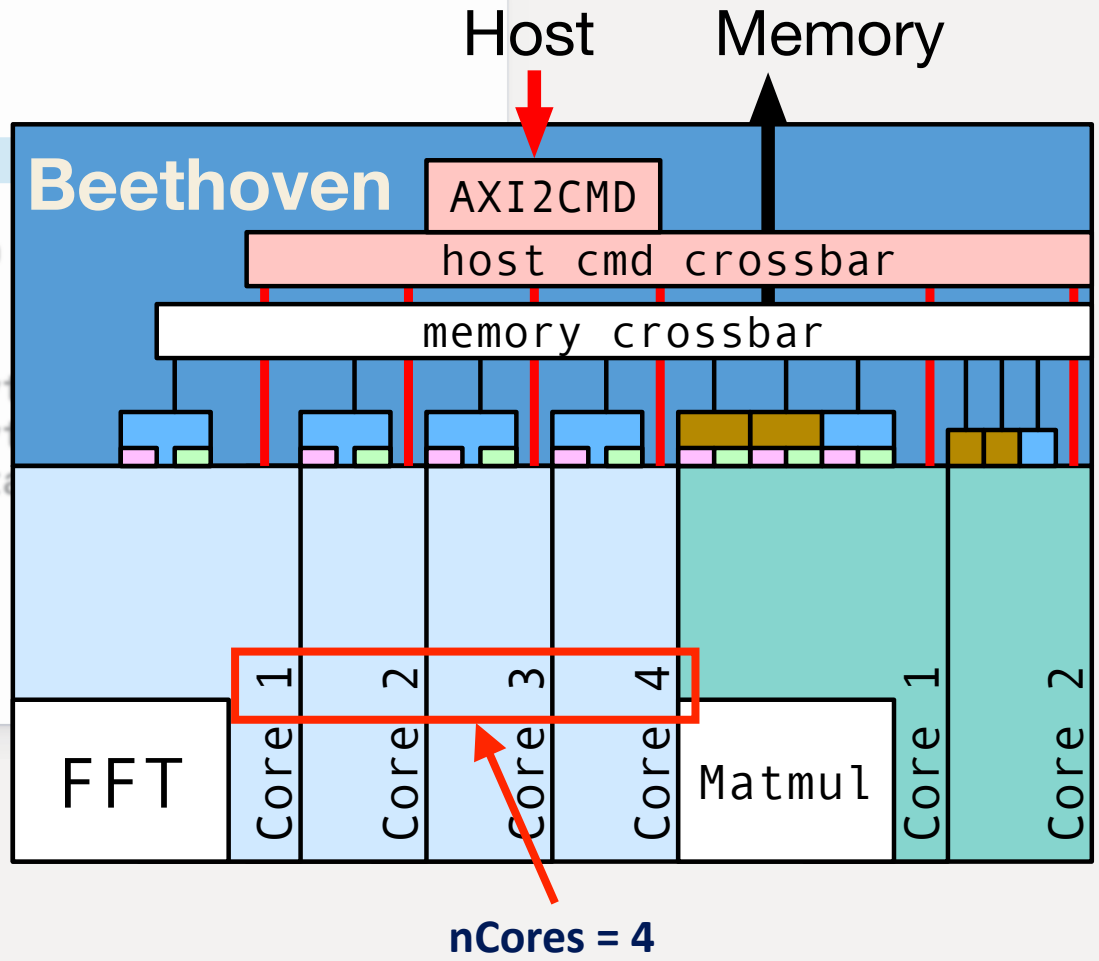
Accelerator Configuration

DotProductConfig.scala

```
1  class DotProductConfig extends AcceleratorConfig({
2    ...
3    List(
4      AcceleratorSystemConfig(
5        nCores = 1,
6        name = "DotProductCore",
7        moduleConstructor = ModuleBuilder(p =>
8            new DotProductCore()(p)),
9        memoryChannelConfig = List(
10         ReadChannelConfig("vec_a", dataBytes = pairBytes),
11         ReadChannelConfig("vec_b", dataBytes = pairBytes),
12         WriteChannelConfig("vec_out", dataBytes = pairBytes)
13       )
14     )
15   )
16 })
```

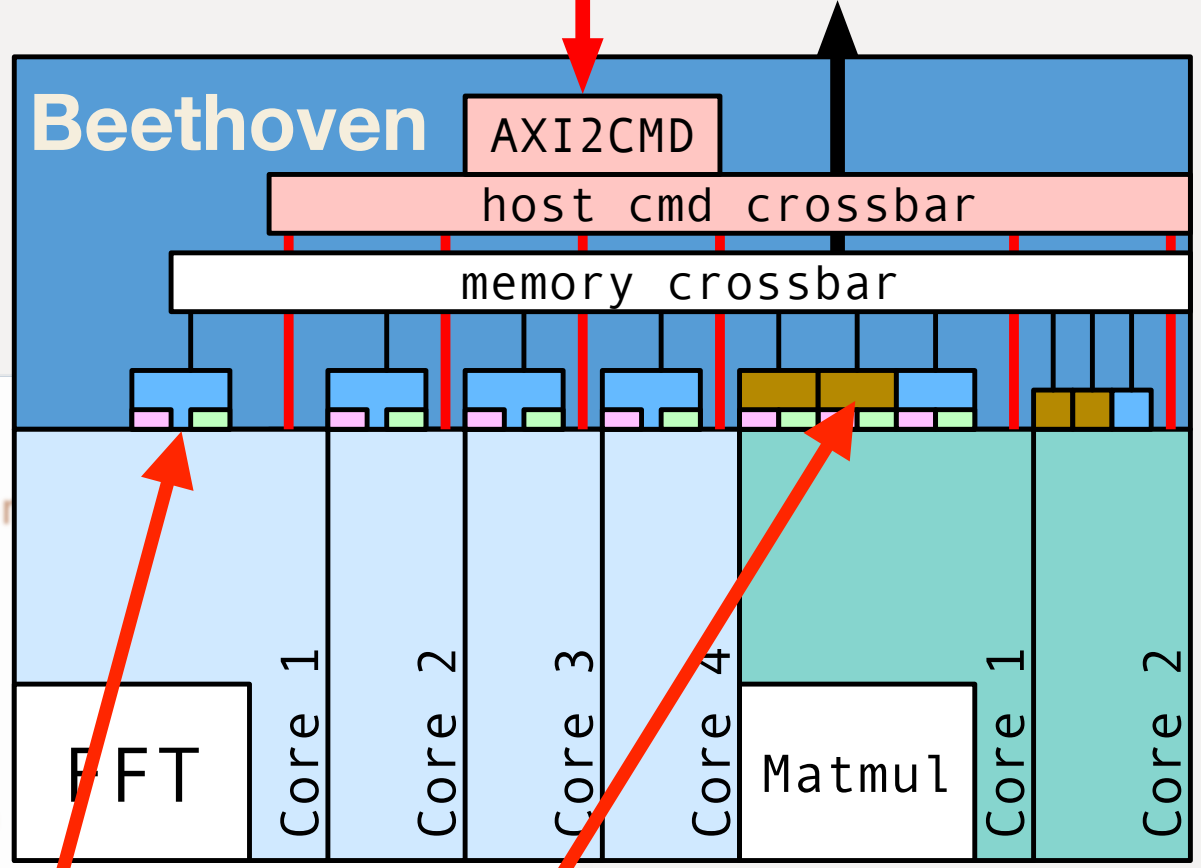
DotProductConfig.scala

```
class DotProductConfig extends AcceleratorConfig({
  ...
  List(
    AcceleratorSystemConfig(
      nCores = 1,
      name = "DotProductCore",
      moduleConstructor = ModuleBuilder(p
          new DotProductCore()(p)),
      memoryChannelConfig = List(
        ReadChannelConfig("vec_a", dataBy
        ReadChannelConfig("vec_b", dataBy
        WriteChannelConfig("vec_out", data
      )
    )
  )
})
```



Number of homogenous cores: INT

Duke

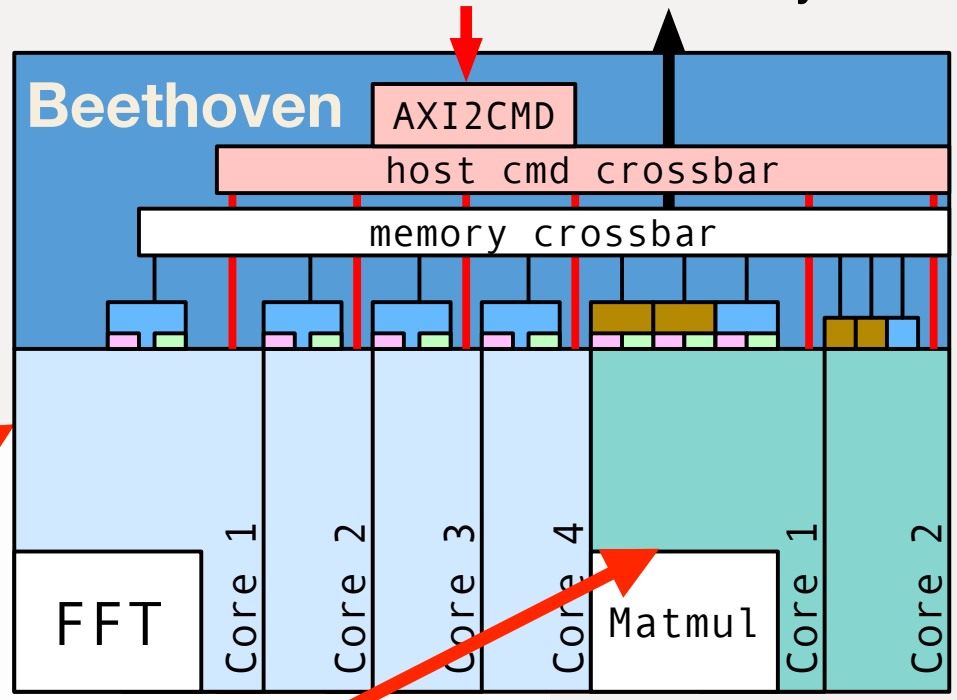


DotProductConfig.scala

```
class DotProductConfig extends Accelerator
  ...
  List(
    AcceleratorSystemConfig(
      nCores = 1,
      name = "DotProductCore",
      moduleConstructor = ModuleBuilder(p
          new DotProductCore()(p)),
      memoryChannelConfig = List(
        ReadChannelConfig("vec_a", dataBytes = pairBytes),
        ReadChannelConfig("vec_b", dataBytes = pairBytes),
        WriteChannelConfig("vec_out", dataBytes = pairBytes)
      )
    )
  )
})
```

Reader/Writer definition:
dataBytes defines the width of the bus

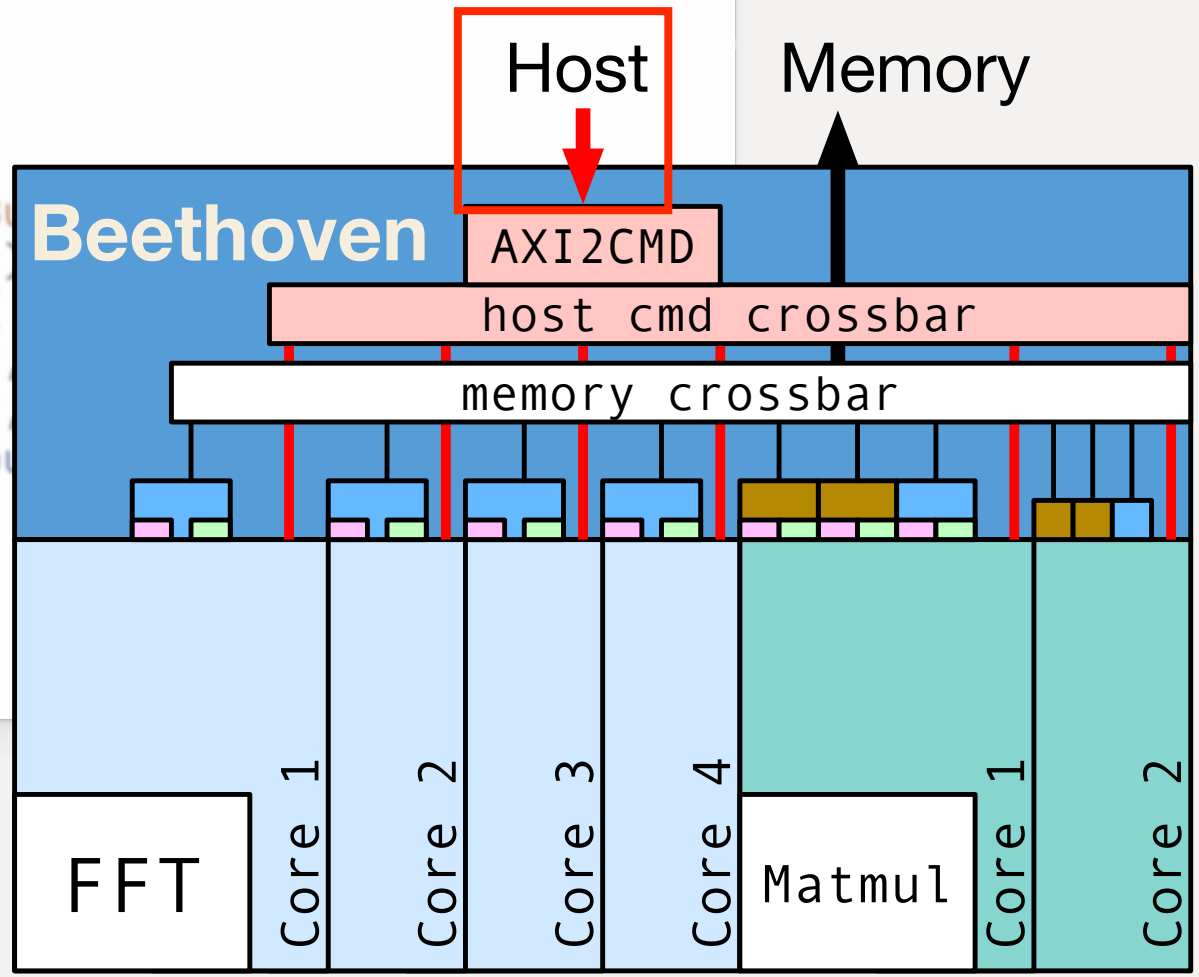
Duke




```
List(
  AcceleratorSystemConfig(
    nCores = 4,
    name = "FFT",
    moduleConstructor = ModuleBuilder(p => new FFT()(p)),
    ...
  ),
  AcceleratorSystemConfig(
    nCores = 2,
    name = "Matmul",
    moduleConstructor = ModuleBuilder(p => new ()(p)),
    ...
  )
)
```

Heterogenous Core Definition

```
class DotProductConfig extends AcceleratorConfig({
  ...
  List(
    AcceleratorSystemConfig(
      nCores = 1,
      name = "DotProductCore",
      moduleConstructor = ModuleBu
          new DotProductCore()(p)
      memoryChannelConfig = List(
        ReadChannelConfig("vec_a"
        ReadChannelConfig("vec_b"
        WriteChannelConfig("vec_ou
      )
    )
  )
})
```



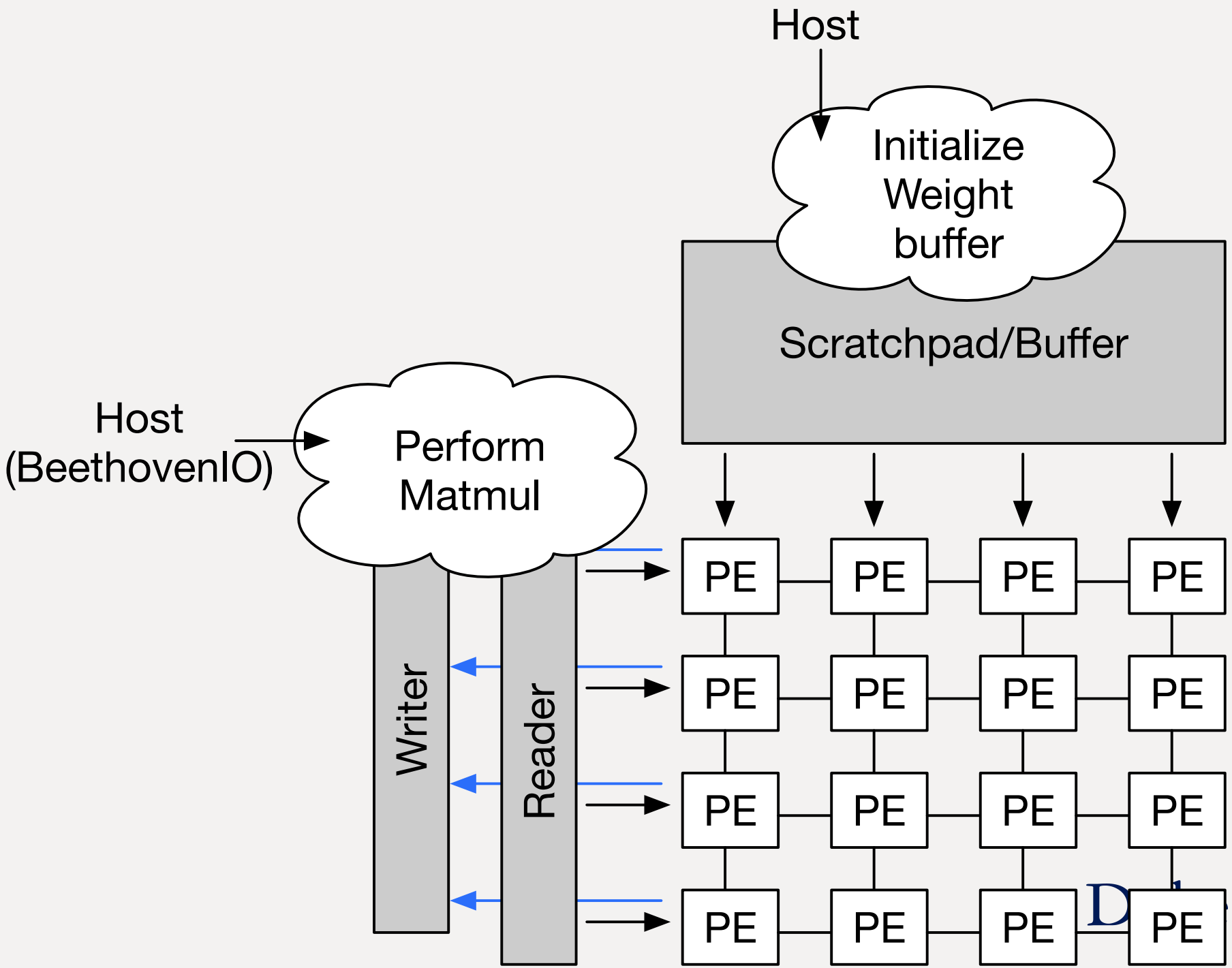
name tells how the host identifies this remote accelerator system

Duke

Your turn

```
.
├── Beethoven.toml
├── README.md
├── build.sbt
├── hw
│   └── src # <----- SystolicArrayConfig.scala
├── project
│   ├── build.properties
│   └── target
├── sw
│   ├── CMakeLists.txt
│   └── systolic_array_tb.cc # <----- Software Testbench
......
```

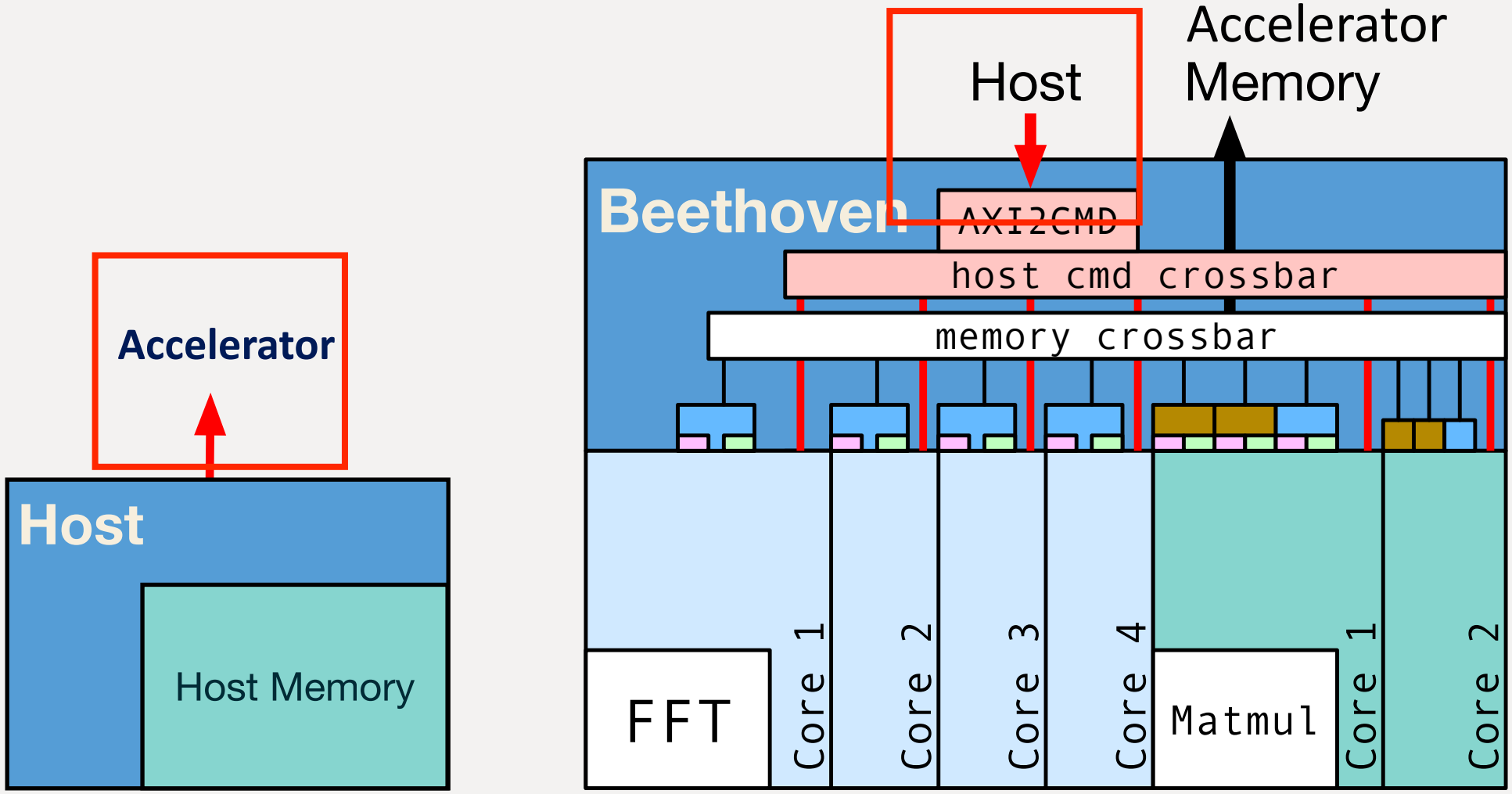
Duke


Host
Initialize
Weight
buffer
Scratchpad/Buffer
Host
(BeethovenIO)
Perform
Matmul
Writer
Reader
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE

Host-Accelerator Interface

Host-Accelerator I/O

Host-Accelerator Interface defines how the host sends command to accelerators and receives responses back.



```
.
├── Beethoven.toml
├── README.md
├── build.sbt
├── hw
│   └── src # <-----  Using Scala : DotProductConfig.scala
│             <-----  Using Verilog: DotProductCmd.scala
├── project
│   ├── build.properties
│   └── target
├── sw
│   ├── CMakeLists.txt
│   └── systolic_array_tb.cc # <----- Software Testbench
.....
```

Duke

Host-Accelerator I/O

- `BeethovenIO` exposes a hardware interface and a software interface for your testbench

Host-Accelerator I/O

- `BeethovenIO` exposes a hardware interface and a software interface for your testbench

core.scala / Cmd.scala — Hardware Side Receive

```
class DotProductCmd(implicit p:Parameters) extend
  AccelCommand("dot_product") {
    val in_addr_a = Address()
    val in_addr_b = Address()
    val length  = UInt(20.W)
}
```

Host-Accelerator I/O

- `BeethovenIO` exposes a hardware interface and a software interface for your testbench

core.scala / Cmd.scala — Hardware Side Receive

```
class DotProductCmd(implicit p:Parameters) extend
  AccelCommand("dot_product") {
    val in_addr_a = Address()
    val in_addr_b = Address()
    val length  = UInt(20.W)
}
```

Host Side Transmit

```
auto resp = DotProductCore::dot_product(0, input_a, input_b, kLength).get();
```

Core Index

Host-Accelerator I/O

- `BeethovenIO` exposes a hardware interface and a software interface for your testbench

core.scala / Cmd.scala Hardware Side Receive

```
class DotProductCmd(implicit p:Parameters) extend
  AccelCommand("dot_product") {
    val in_addr_a = Address()
    val in_addr_b = Address()
    val length  = UInt(20.W)
}
```

Address Type: Address widths may change from platform to platform, abstraction hides width

Duke

Host-Accelerator I/O

- `BeethovenIO` exposes a hardware interface and a software interface for your testbench

dot_product_tb.cc Hardware Side Transmit

```
class DotProductResp extends AccelResponse("dot_product_resp") {
  val result = UInt(64.W)
}
```

Host-Accelerator I/O

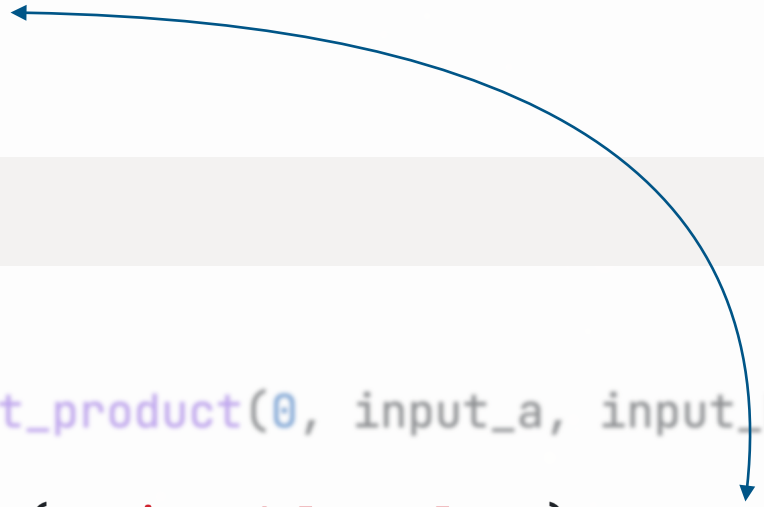
- `BeethovenIO` exposes a hardware interface and a software interface for your testbench

core.scala / Cmd.scala — Hardware Side Transmit

```
class DotProductResp extends AccelResponse("dot_product_resp") {
  val result = UInt(64.W)
}
```

dot_product_tb.cc — Host Side Receive

```
auto resp = DotProductCore::dot_product(0, input_a, input_b, kLength).get();

std::printf("Result is: %llu", (unsigned long long)resp.result);
```



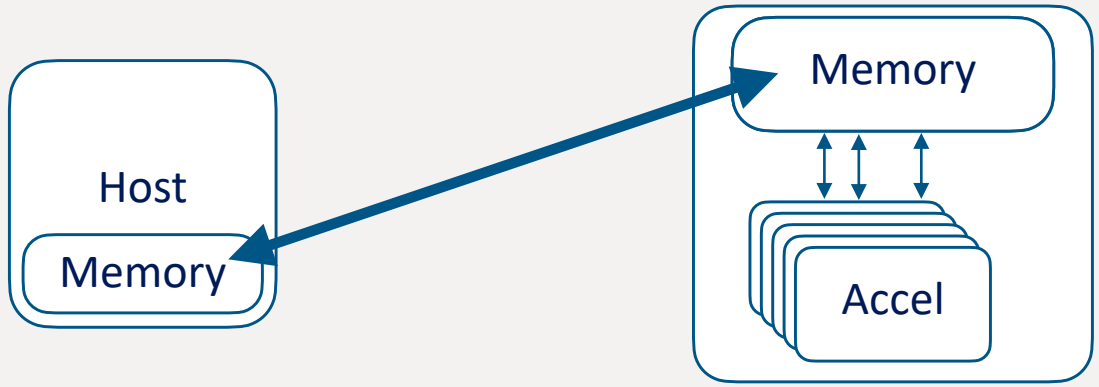
Duke

Host-Accelerator I/O

Beethoven also provides DMA helpers for larger data transmission from host to accelerators

dot_product_tb.cc — Software Side DMA

```
auto input_a = handle.malloc(sizeof(uint32_t) * kLength);
auto input_b = handle.malloc(sizeof(uint32_t) * kLength);
////////////////////////////////////////
// Host Write Data to input a and input b //
////////////////////////////////////////
handle.copy_to_fpga(input_a);
handle.copy_to_fpga(input_b);
```

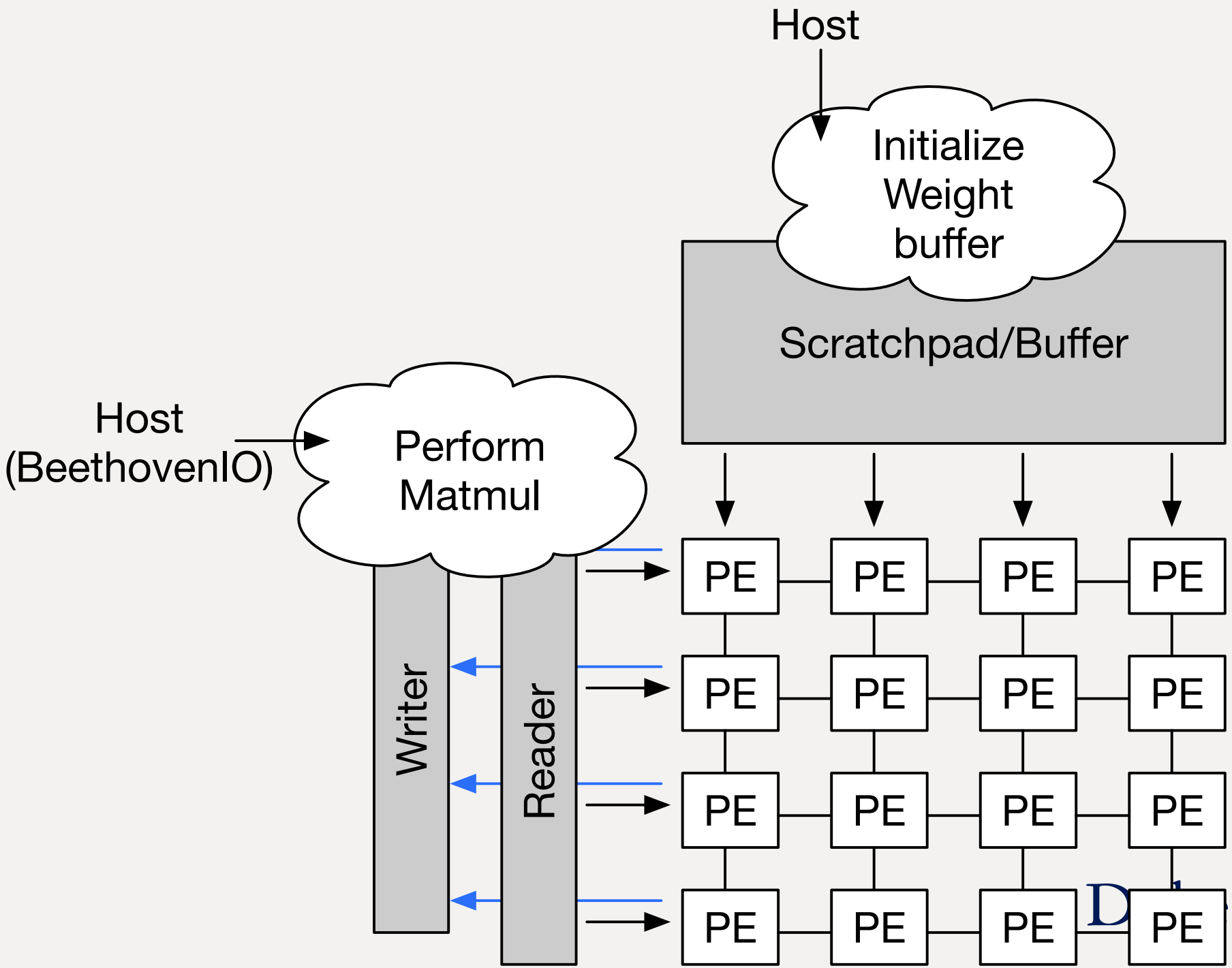


Duke

```
.
├── Beethoven.toml
├── README.md
├── build.sbt
├── hw
│   └── src # <-----  Using Scala : DotProductConfig.scala
│             <-----  Using Verilog: DotProductCmd.scala
├── project
│   ├── build.properties
│   └── target
├── sw
│   ├── CMakeLists.txt
│   └── systolic_array_tb.cc # <----- Software Testbench
.....
```

Your turn

Duke

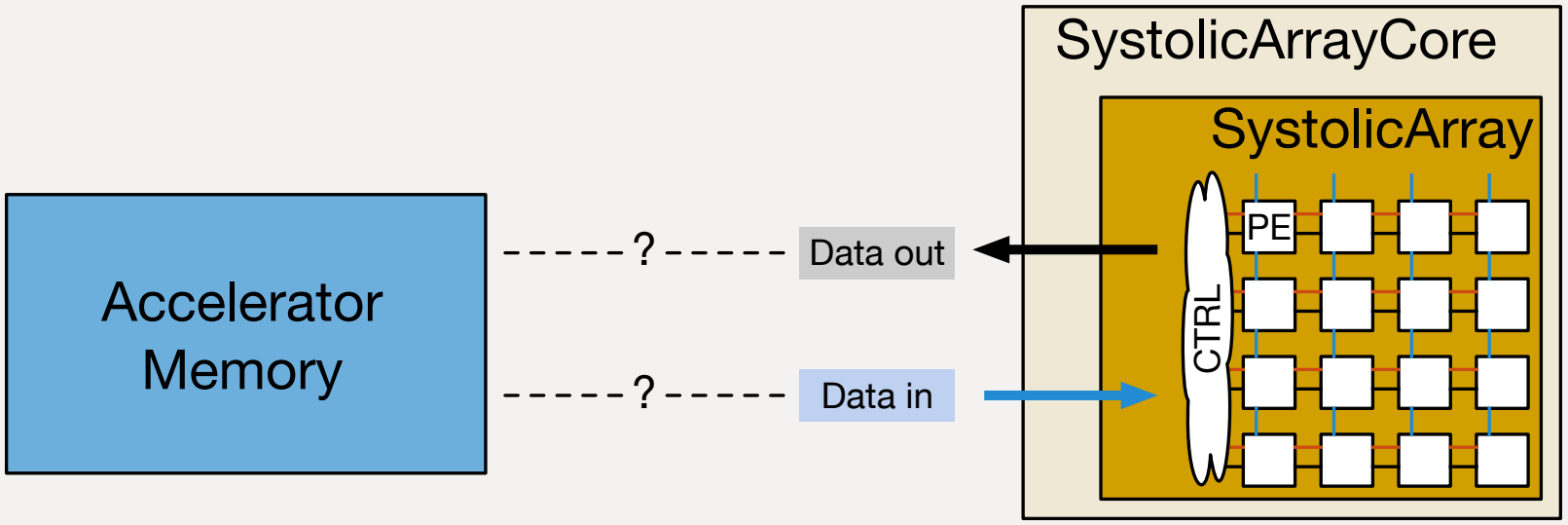

Host
Initialize
Weight
buffer
Scratchpad/Buffer
Host
(BeethovenIO)
Perform
Matmul
Writer
Reader
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE

Memory Interface

Read/Write Channels

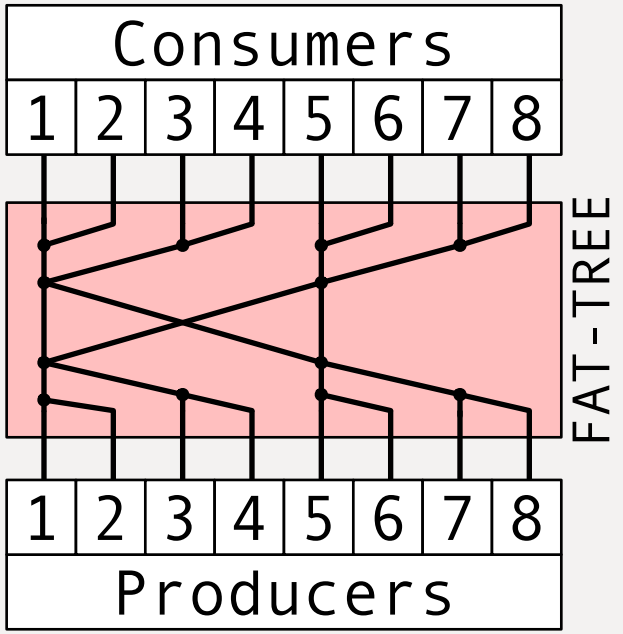
Challenges for Memory Access

- How do you move data from accelerator memory (DRAM) to your accelerator?



Duke

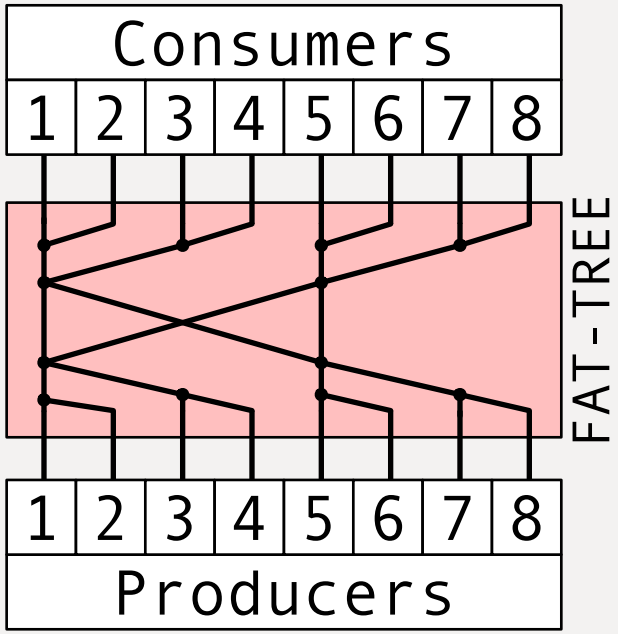
Challenges for Memory Access



Interconnect design is tricky!

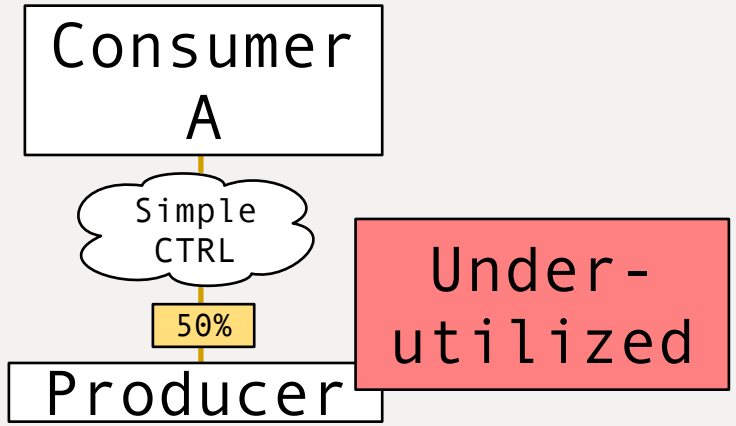
NIC Design

Challenges for Memory Access



Interconnect design is tricky!

NIC Design

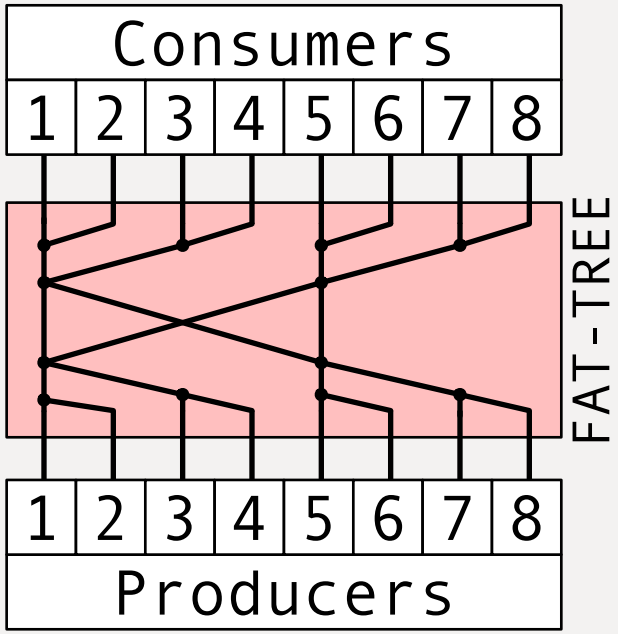


Under-utilized

Careful design is required to achieve high-performance

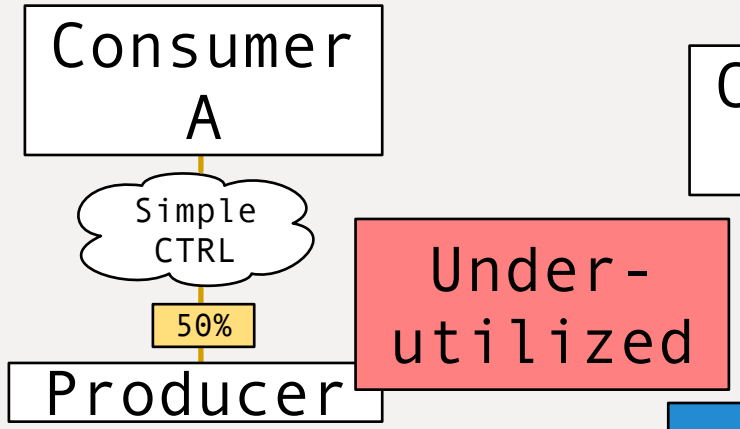
High-Performance

Challenges for Memory Access



Interconnect design is tricky!

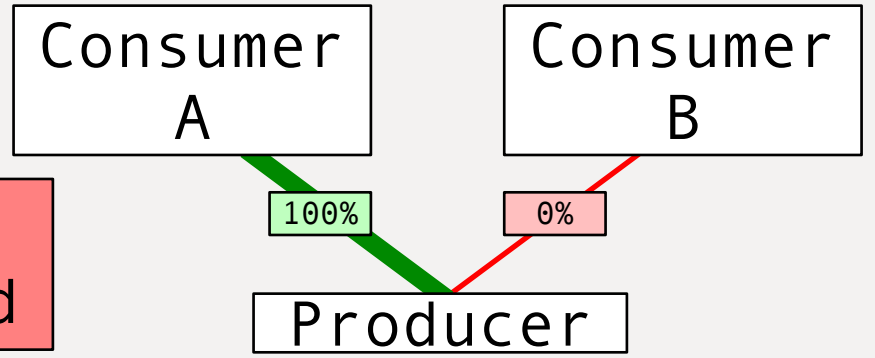
NIC Design



Under-utilized

Careful design is required to achieve high-performance

High-Performance



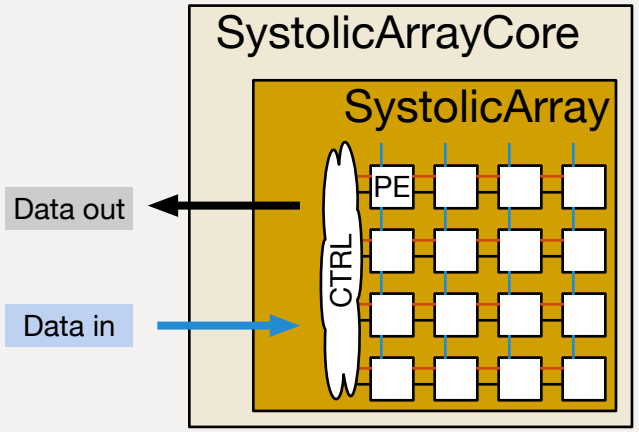
Both A, B want 100% of bandwidth, requires control for fairness

Contention/Fairness

Duke

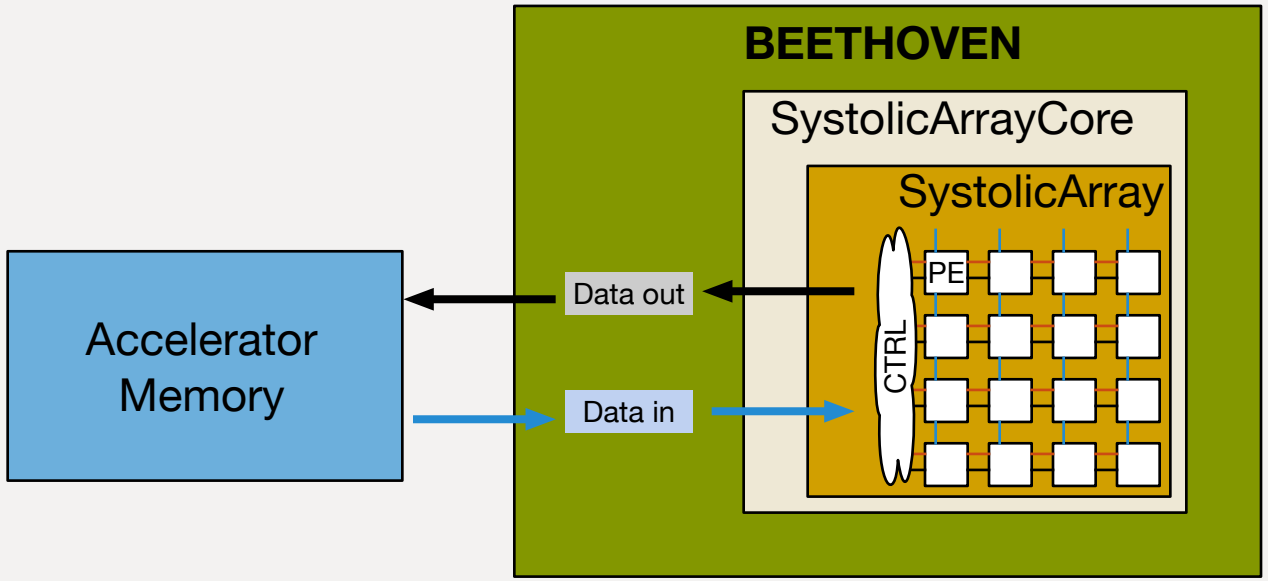
Integration of Memory Streams

- Within the design of your accelerator core, you have data streams to/from memory



Integration of Memory Streams

- Within the design of your accelerator core, you have data streams to/from memory

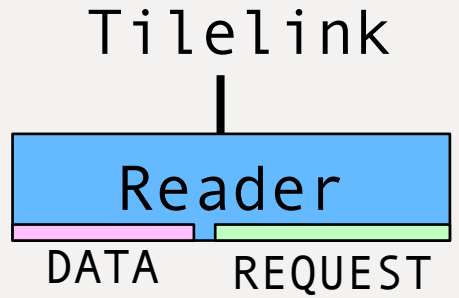


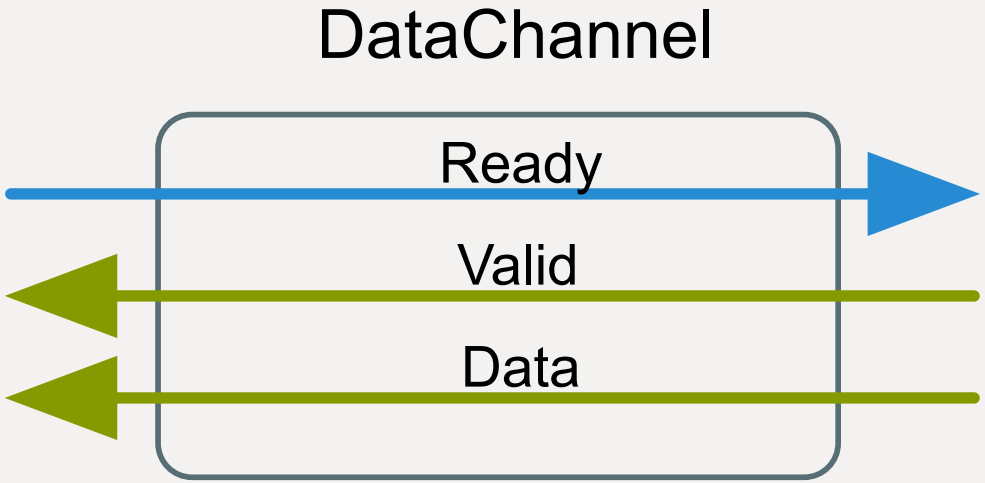
Beethoven provides Memory Access Abstractions in RTL to ease these challenges

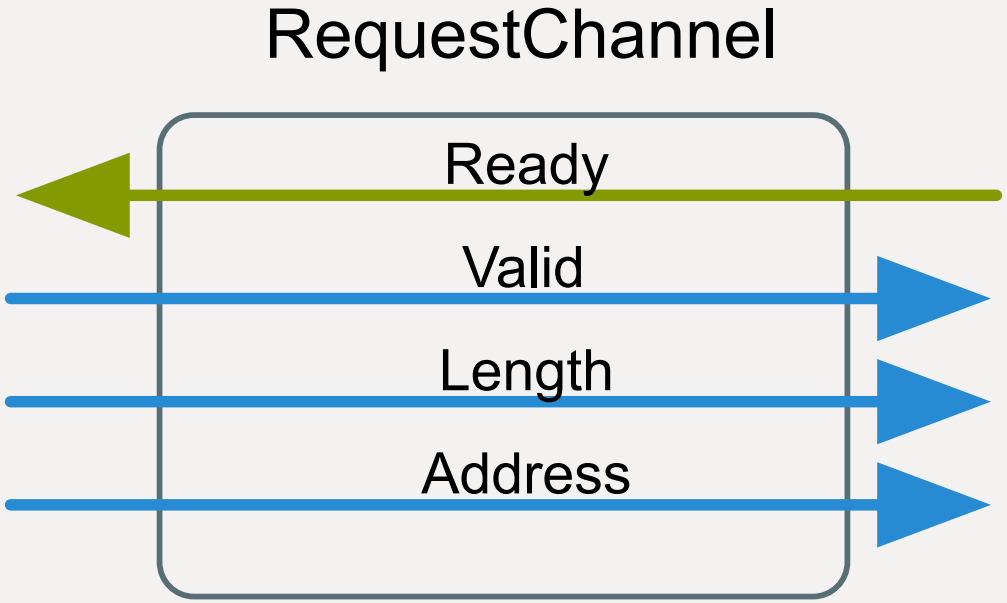
Duke

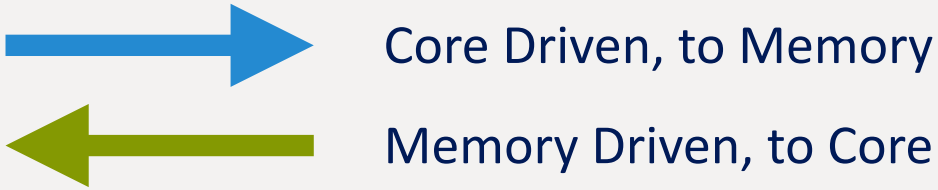
Memory Streams - Readers

- Readers contain a **data channel** and a **request channel**





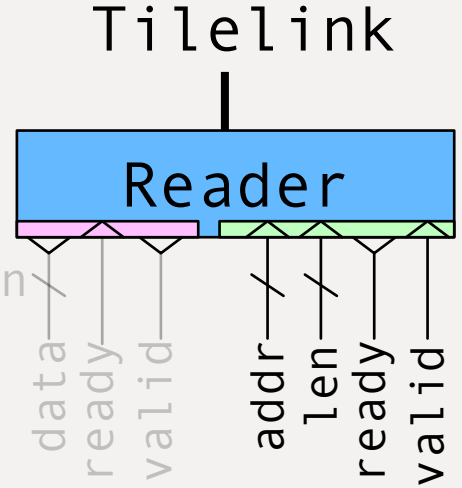




Duke

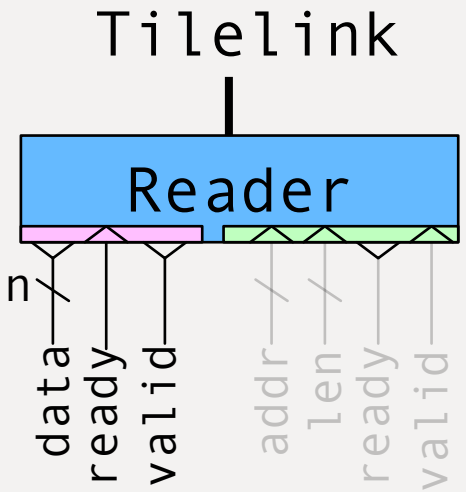
Memory Streams - Readers

- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data

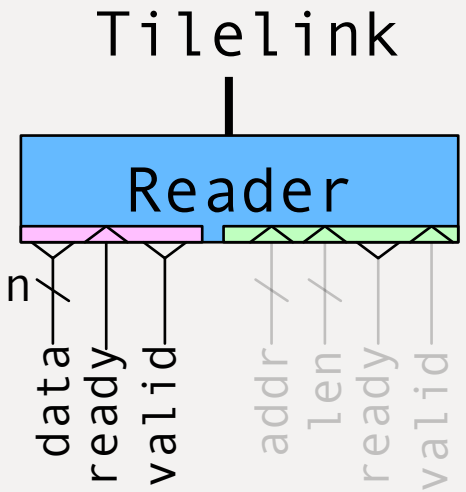


Memory Streams - Readers

- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data
  - Data: Handshake + data item



Memory Streams - Readers

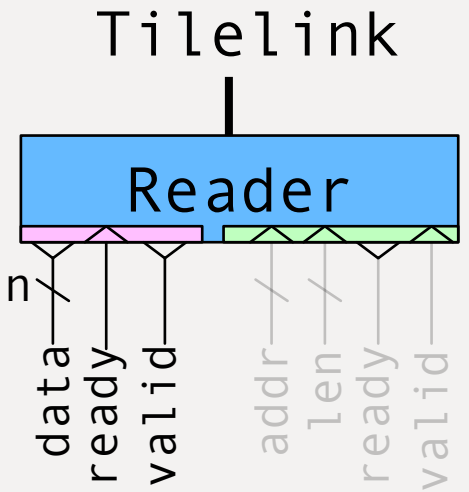


- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data
  - Data: Handshake + data item

Inside RTL: "Get" a reader

```
class MyAccel extends AcceleratorCore {
  val io = BeethovenIO(…)
  val myReader: ReaderModuleChannel =
    getReader(“readA”)
  // set up request channel
  myReader.requestChannel := ...
  // set up data channel
  myReader.dataChannel := ...
```

Memory Streams - Readers



- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data
  - Data: Handshake + data item

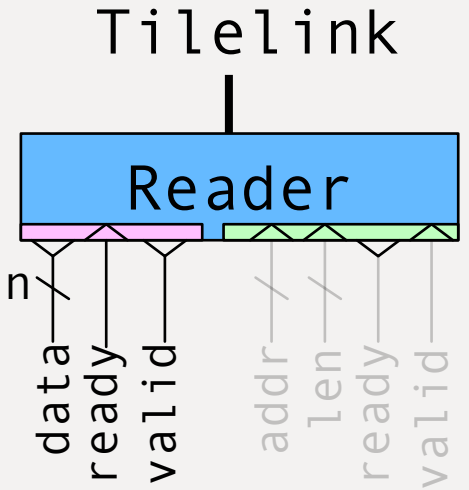
Inside RTL: "Get" a reader

```
class MyAccel extends AcceleratorCore {
  val io = BeethovenIO(…)
  val myReader: ReaderModuleChannel =
    getReader(“readA”)
  // set up request cha
  myReader.requestChann
  // set up data channe
  myReader.dataChannel
```

The name of the reader links this "get" to a reader in your configuration

Duke

Memory Streams - Readers



- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data
  - Data: Handshake + data item

Inside RTL: "Get" a reader

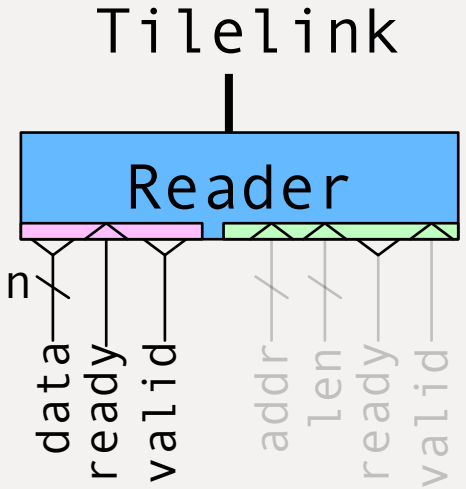
```
class MyAccel extends AcceleratorCore {
  val io = BeethovenIO(…)
  val myReader: ReaderModuleChannel =
    getReader("readA")
  // set up request channel
  myReader.requestChannel := ...
  // set up data channel
  myReader.dataChannel := ...
```

Use the reader like a normal hardware module

Duke

Memory Streams - Readers

- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data
  - Data: Handshake + data item



Inside RTL: "Get" a reader

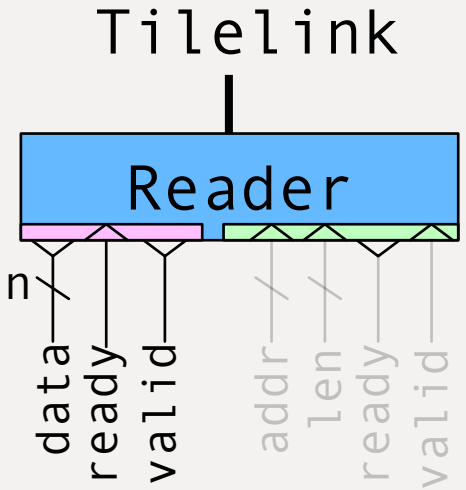
```
class MyAccel extends AcceleratorCore {
  val io = BeethovenIO(…)
  val myReader: ReaderModuleChannel =
    getReader("readA")
  // set up request channel
  myReader.requestChannel := ...
  // set up data channel
  myReader.dataChannel := ...
```

Inside Accel. Configuration: Declare Reader

```
class MyAccelConfig extends AcceleratorConfig(
 AcceleratorSystemConfig(
  nCores = 1,
  name = "myAccelSystem",
  moduleConstructor =
   ModuleBuilder(p => new MyAccel()(p)),
  memoryChannelConfig = List(
   ReadChannelConfig("readA", dataBytes = 4),
   ReadChannelConfig("readB", dataBytes = 4),
   WriteChannelConfig("write", dataBytes = 4)
)))
```

Memory Streams - Readers

- Readers contain a **data channel** and a **request channel**
  - Request: Handshake + Addr & length (in bytes) for a contiguous stream of data
  - Data: Handshake + data item



Inside RTL: "Get" a reader

```
class MyAccel extends AcceleratorCore {
  val io = BeethovenIO(…)
  val myReader: ReaderModuleChannel =
    getReader("readA")
  // set up request channel
  myReader.requestChannel := ...
```

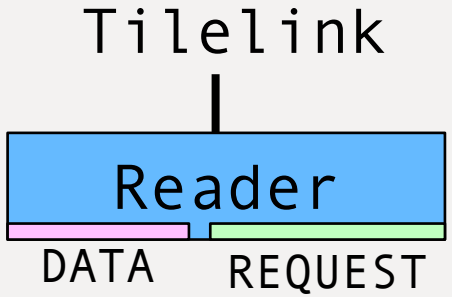
Inside Accel. Configuration: Declare Reader

```
class MyAccelConfig extends AcceleratorConfig(
 AcceleratorSystemConfig(
  nCores = 1,
  name = "myAccelSystem",
  moduleConstructor =
   ModuleBuilder(p => new MyAccel()(p)),
  memoryChannelConfig = List(
   ReadChannelConfig("readA", dataBytes = 4),
   ReadChannelConfig("readB", dataBytes = 4),
   WriteChannelConfig("write", dataBytes = 4)
)))
```

Accelerator Configuration tells Beethoven about high-level SoC details. Your Accelerator interacts with the elaborated SoC (inc. Readers)

Memory Streams - Writer

- Writers are the same as readers, except for direction of the data channel

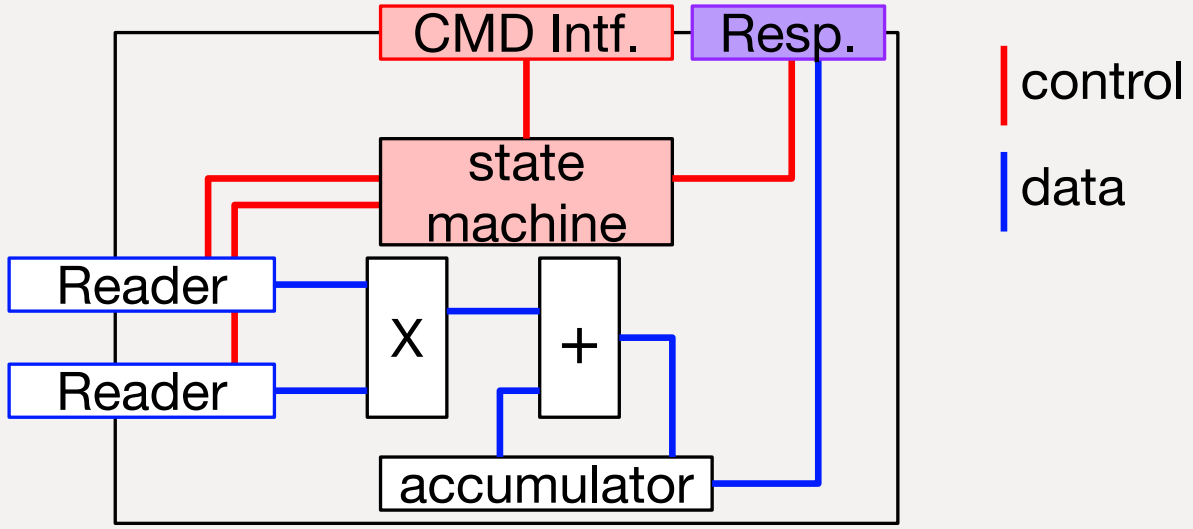



```
class MyAccel extends AcceleratorCore {
  val io = BeethovenIO(…)
  val myWriter: WriterModuleChannel =
    getWriter("write")
  // set up request channel
  myWriter.requestChannel := ...
  // set up data channel
  myWriter.dataChannel := ...
```

```
class MyAccelConfig extends AcceleratorConfig(
 AcceleratorSystemConfig(
  nCores = 1,
  name = "myAccelSystem",
  moduleConstructor =
   ModuleBuilder(p => new MyAccel()(p)),
  memoryChannelConfig = List(
   ReadChannelConfig("readA", dataBytes = 4),
   ReadChannelConfig("readB", dataBytes = 4)
   WriteChannelConfig("write", dataBytes = 4)
)))
```

Dot Product Example

- Must stream in input vectors
- Write one element back to host - no output streaming

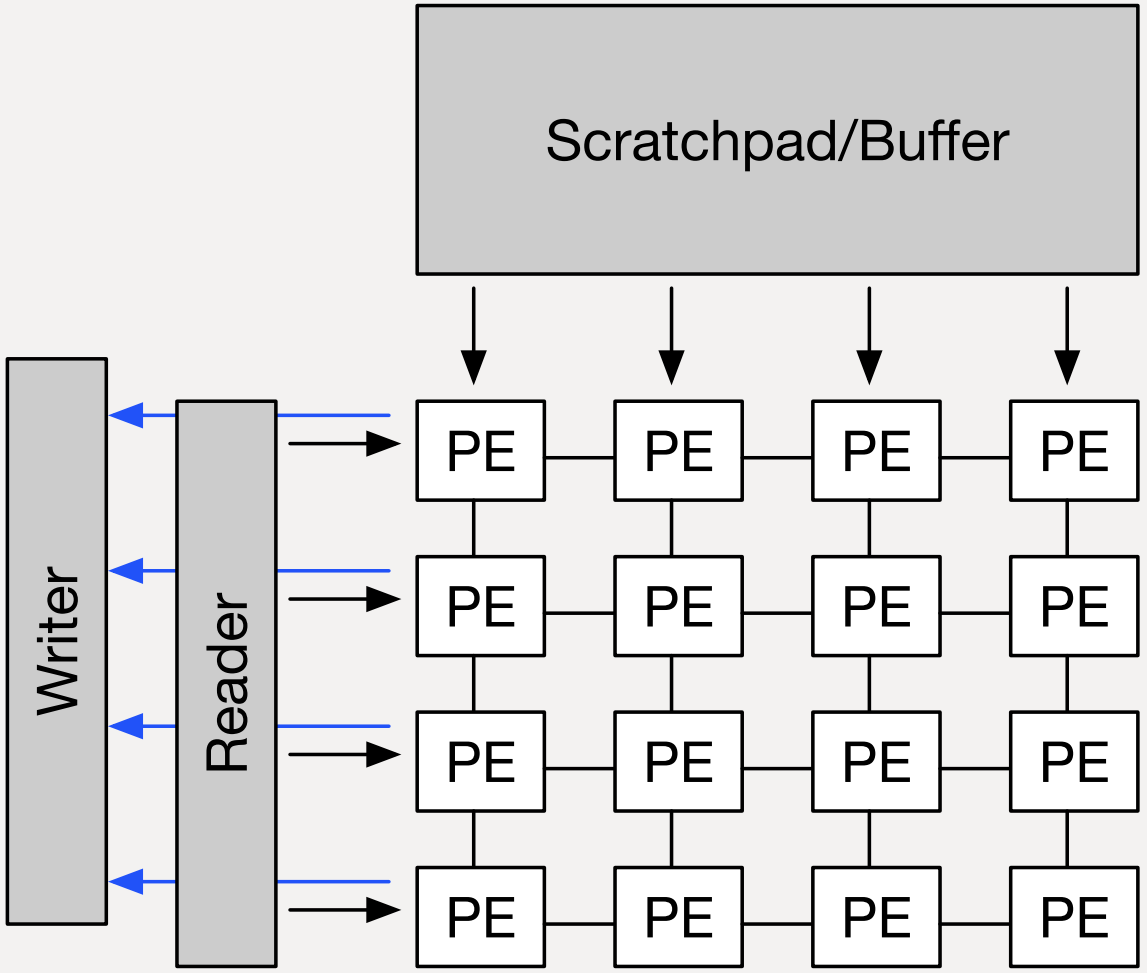


VS Code Dot-Product Example

Duke

Systolic Array Example

- Must stream in input activations
- Must stream out output activations



VS Code Systolic Array Hands-On

Duke

On-Chip Memory Abstractions

Scratchpads and Memories

Memories - Challenges

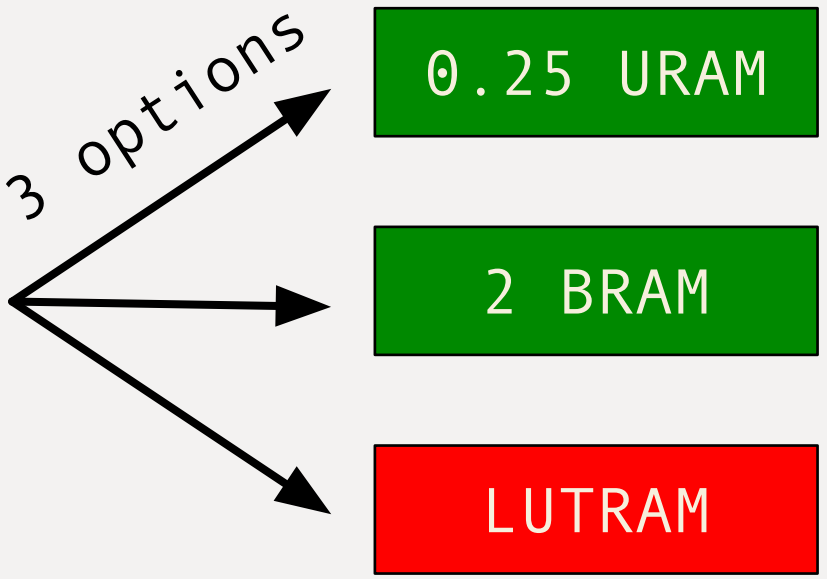
- Memory inference can be tricky! **(Especially in Chisel)**
- Using FPGA memories correctly require best practices (e.g., Verilog Annotations)
- Resources are limited

Chisel

```
class MyModule extends Module {
  val my_bram =
    Reg(Vec(1024, UInt(64.W))
```

Verilog

```
module MyModule (...)
  ...
  reg [63:0] my_bram [0:1023];
  ...
```



Duke

Memories - Challenges

- Memory inference can be tricky! **(Especially in Chisel)**
- Using FPGA memories correctly require best practices (e.g., Verilog Annotations)
- Resources are limited

Does not compile to 2D register

```
reg [63:0] my_bram_0;
reg [63:0] my_bram_1;
reg [63:0] my_bram_2;
...
reg [63:0] my_bram_1023;
```

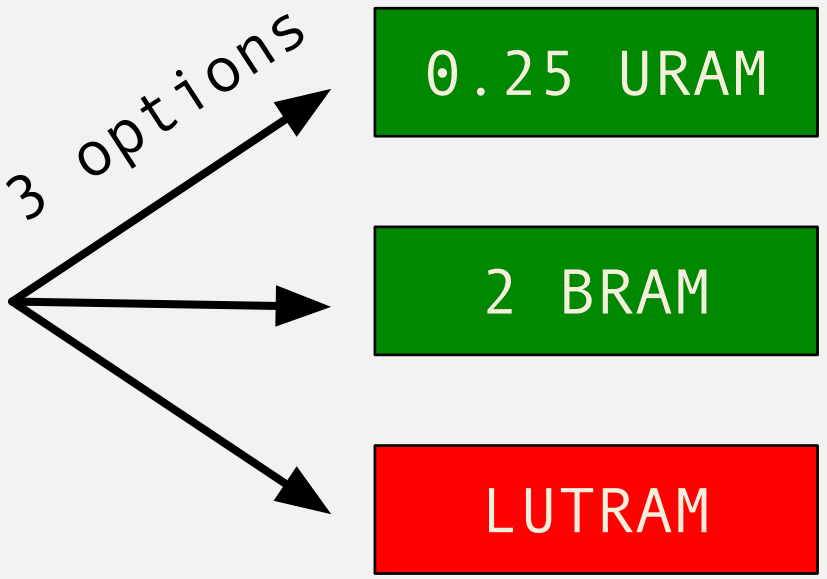
Will likely infer to LUTS

Chisel

```
class MyModule extends Module {
  val my_bram =
    Reg(Vec(1024, UInt(64.W))
```

Verilog

```
module MyModule (...)
  ...
  reg [63:0] my_bram [0:1023];
  ...
```



Duke

Memories - Interfaces

- BRAM vs URAM vs SRAM(ASIC)

Memories - Interfaces

- BRAM vs URAM vs SRAM(ASIC)
    - Single/Dual-Ported memories

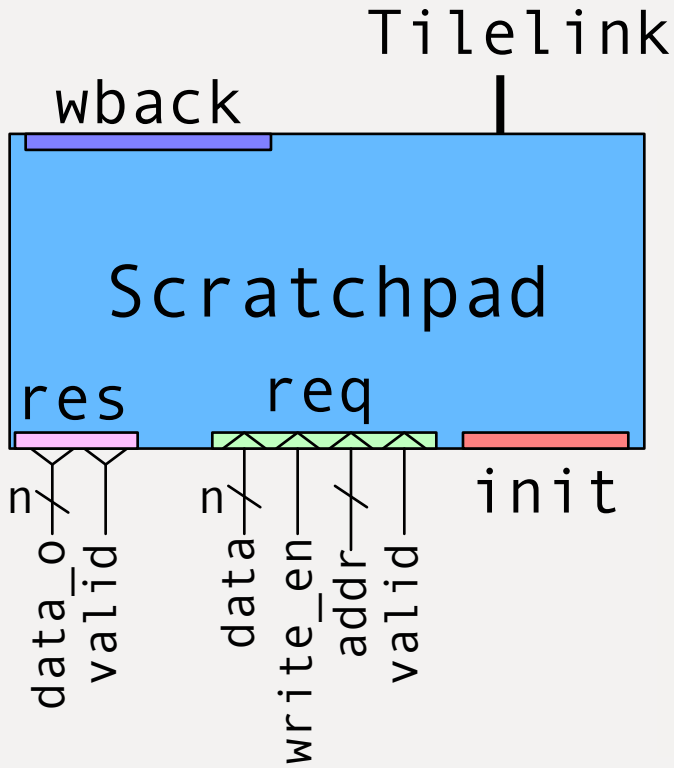
Memories - Interfaces

- BRAM vs URAM vs SRAM(ASIC)
  - Single/Dual-Ported memories
  - Read-Write, Read, Write ports

Memories - Interfaces

- BRAM vs URAM vs SRAM(ASIC)
  - Single/Dual-Ported memories
  - Read-Write, Read, Write ports
  - Different speeds/capacities
  - ASIC SRAMs: PPA Tradeoffs

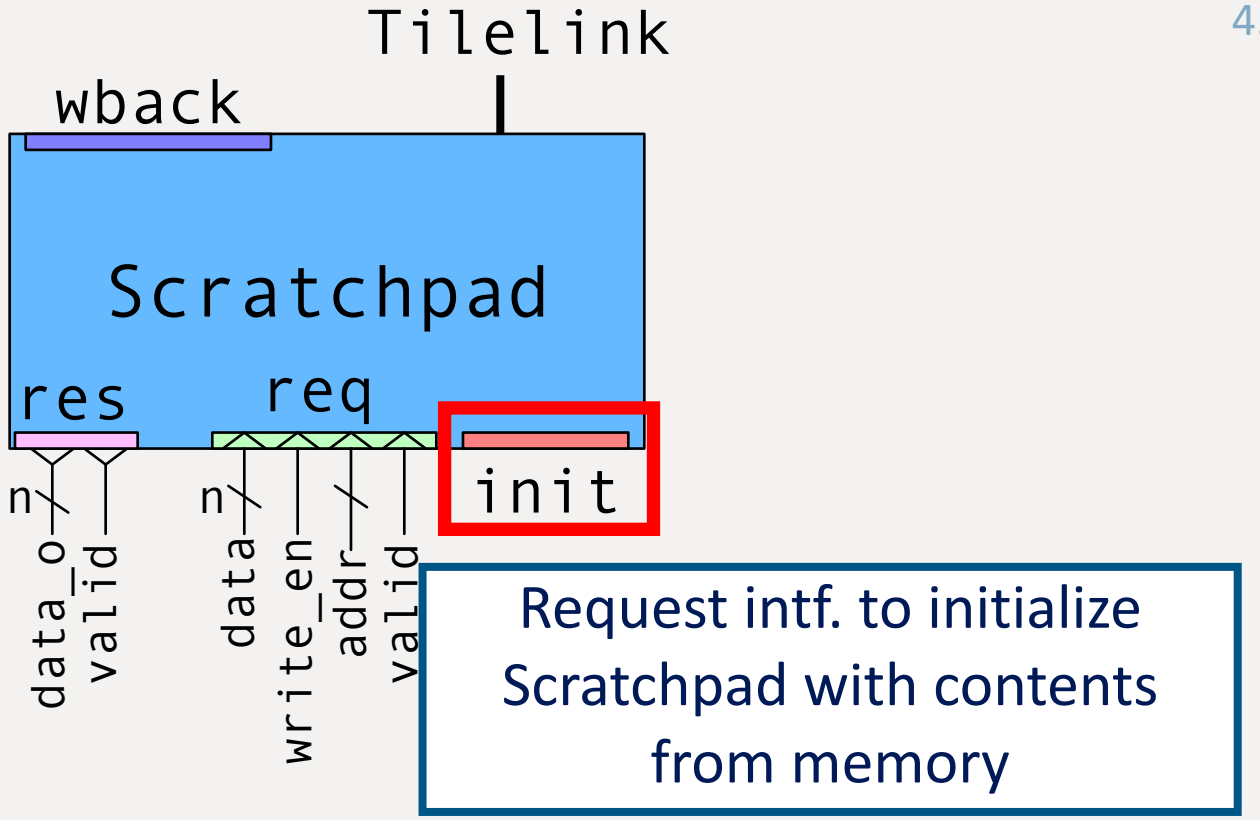
Memories - Interfaces



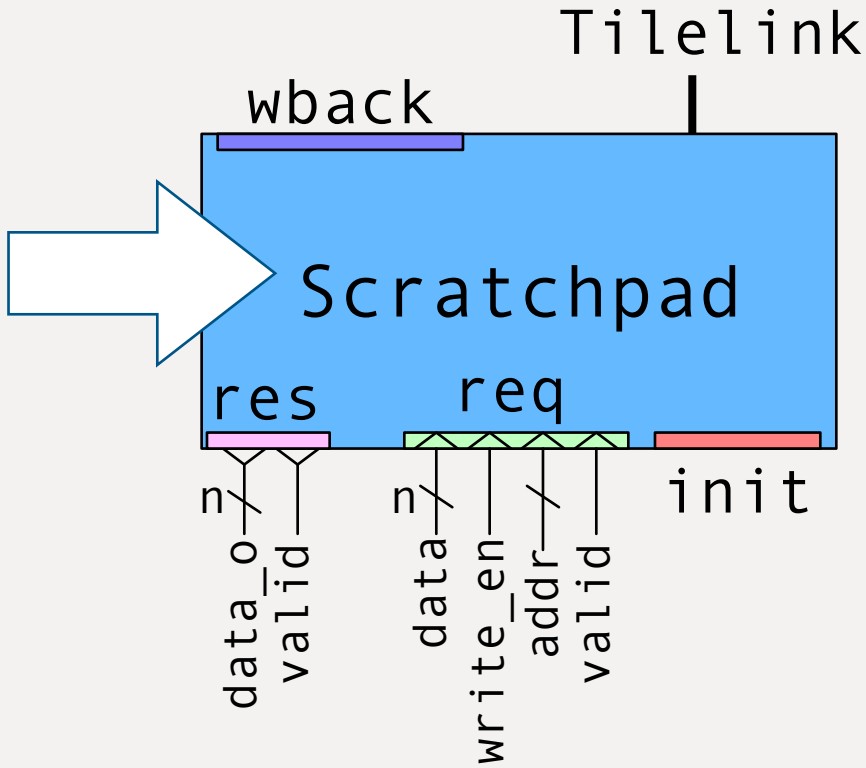
- BRAM vs URAM vs SRAM(ASIC)
  - Single/Dual-Ported memories
  - Read-Write, Read, Write ports
  - Different speeds/capacities
  - ASIC SRAMs: PPA Tradeoffs
- Beethoven Offerings:
  - **Scratchpad memories (Beethoven-backend-managed)**
    - Gives typical SRAM interfaces with additional utilities (init, writeback)

Memories - Interfaces

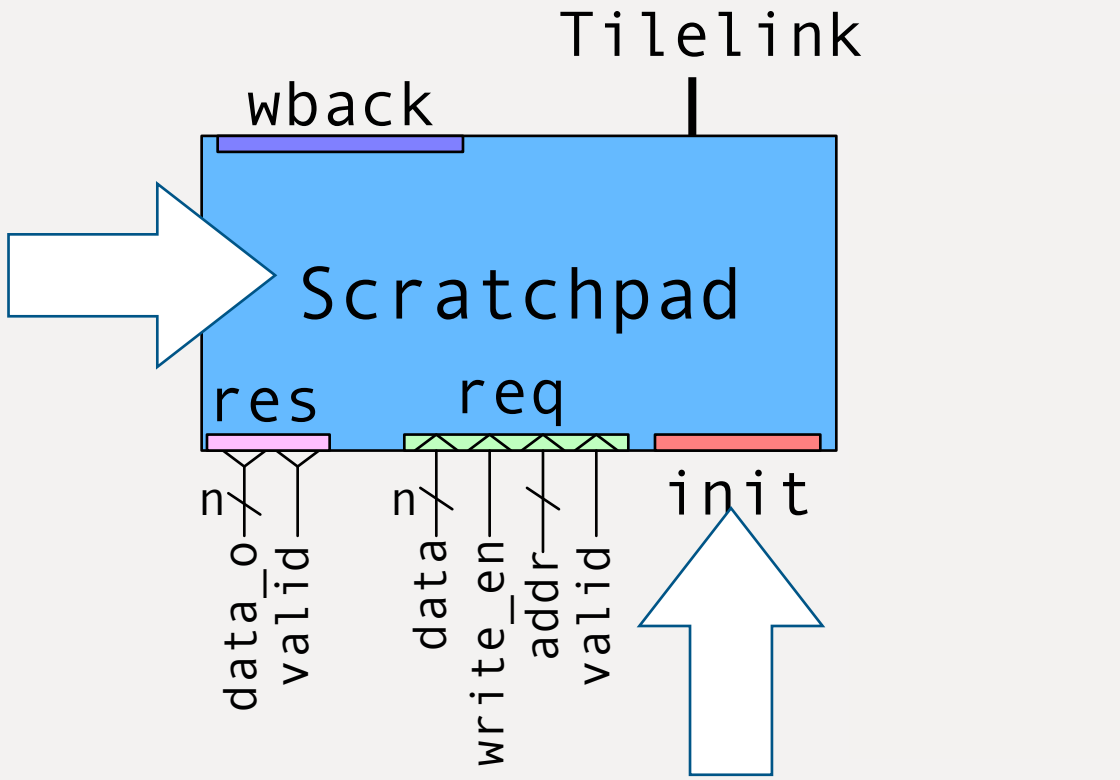
- BRAM vs URAM vs SRAM(ASIC)
  - Single/Dual-Ported memories
  - Read-Write, Read, Write ports
  - Different speeds/capacities
  - ASIC SRAMs: PPA Tradeoffs
- Beethoven Offerings:
  - **Scratchpad memories (Beethoven-backend-managed)**
    - Gives typical SRAM interfaces with additional utilities (init, writeback)



```
0x1030: 00 01 02 03 04 05 06 07
0x1038: 08 09 0a 0b 0c 0d 0e 0f
0x1040: 10 11 12 13 de ad be ef
0x1048: ca fe fe ca 39 a3 93 44
```



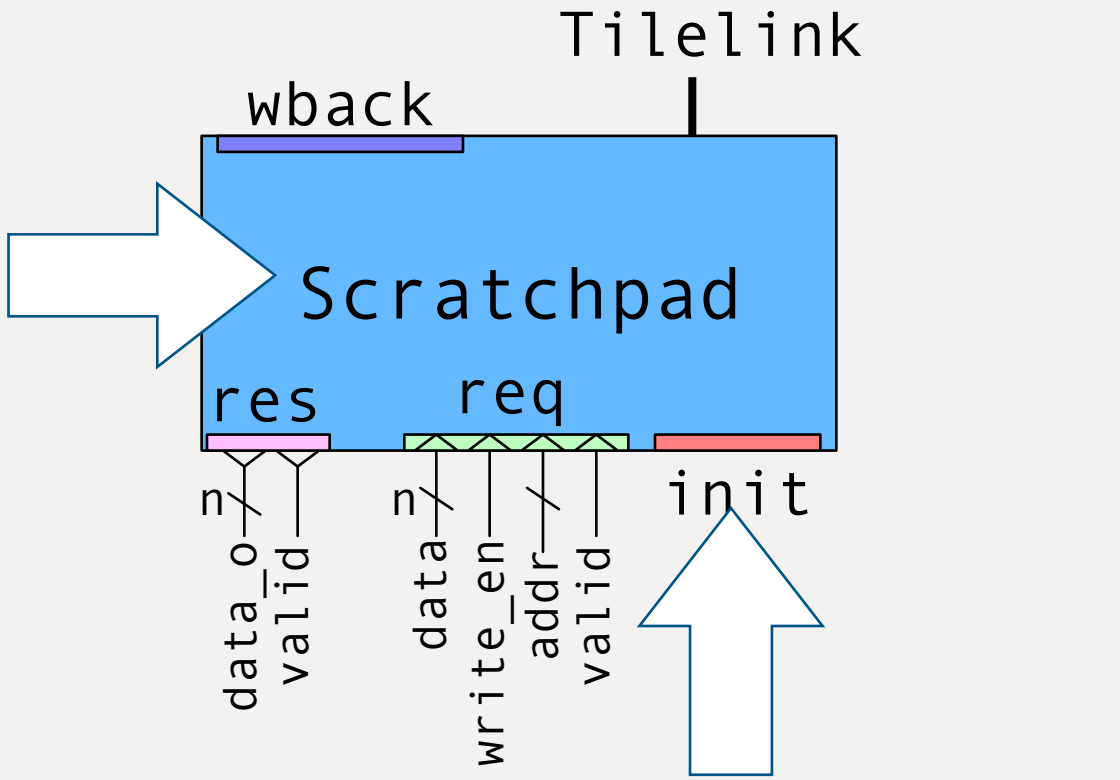
```
0x1030: 00 01 02 03 04 05 06 07
0x1038: 08 09 0a 0b 0c 0d 0e 0f
0x1040: 10 11 12 13 de ad be ef
0x1048: ca fe fe ca 39 a3 93 44
```




```
address: 0x1030
len    : 0x20
```

2-byte datums

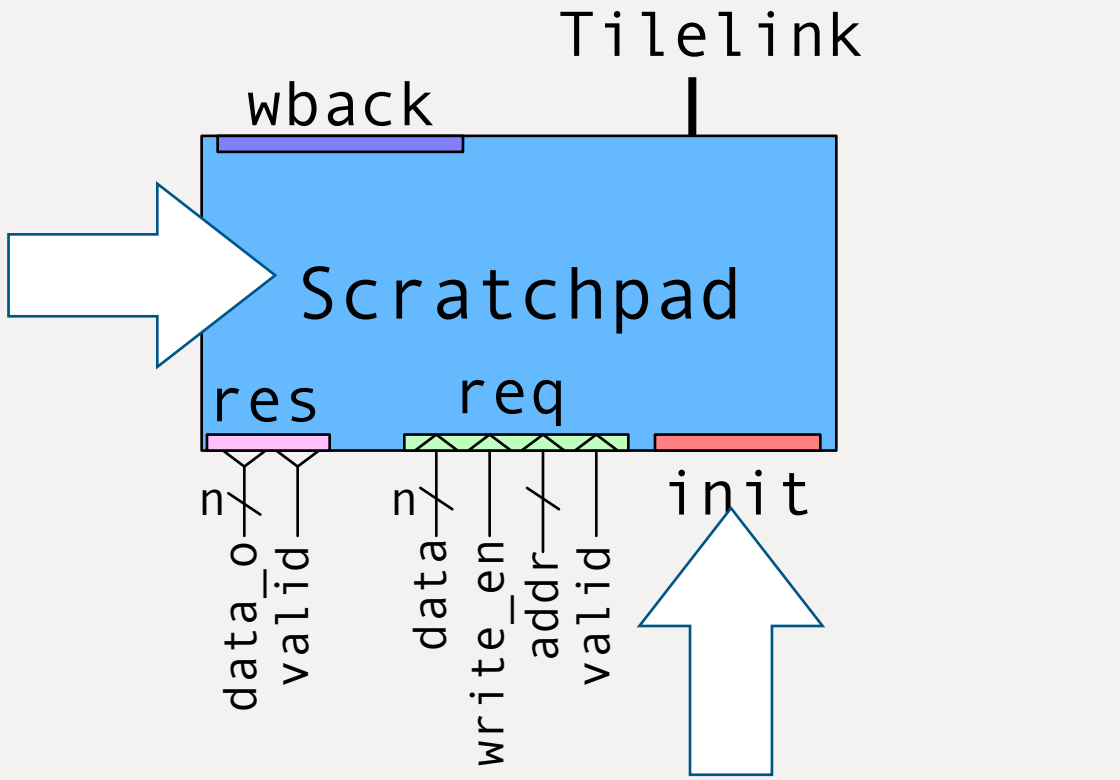
```
0x1030: 00 01 02 03 04 05 06 07
0x1038: 08 09 0a 0b 0c 0d 0e 0f
0x1040: 10 11 12 13 de ad be ef
0x1048: ca fe fe ca 39 a3 93 44
```




```
address: 0x1030
len    : 0x20
```

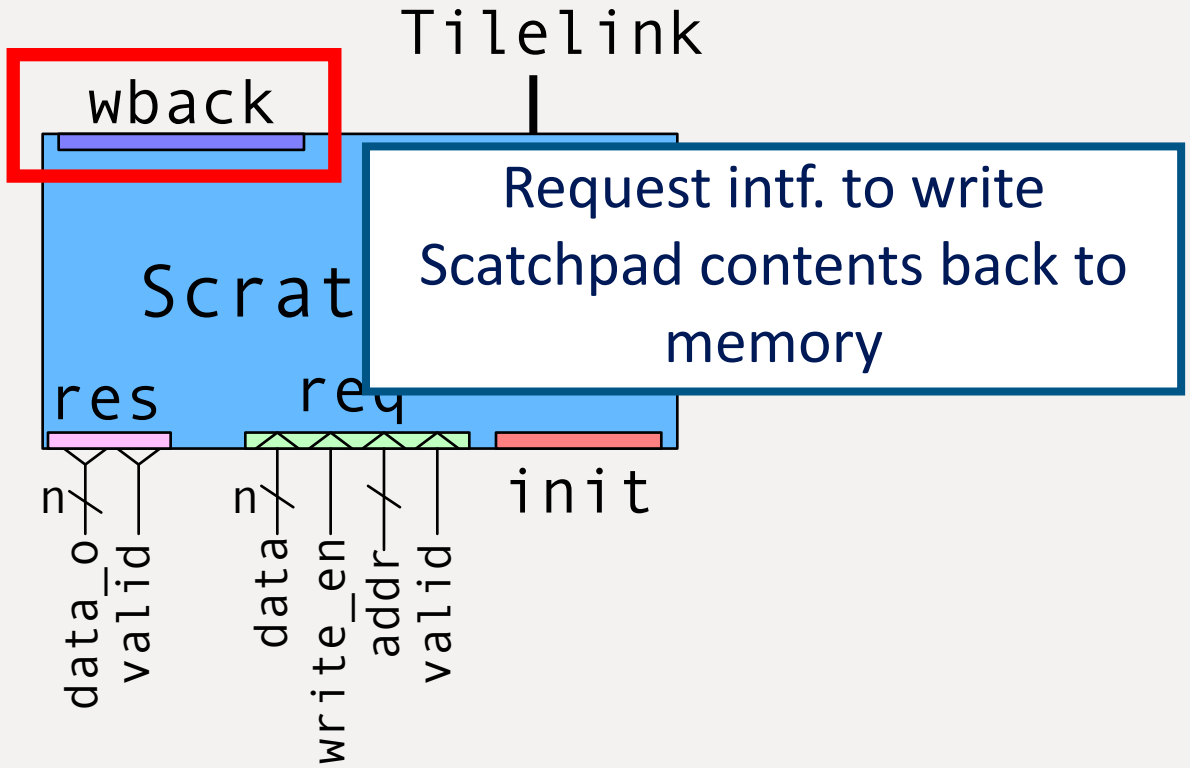
4-byte datums

```
0x1030: 00 01 02 03 04 05 06 07
0x1038: 08 09 0a 0b 0c 0d 0e 0f
0x1040: 10 11 12 13 de ad be ef
0x1048: ca fe fe ca 39 a3 93 44
```



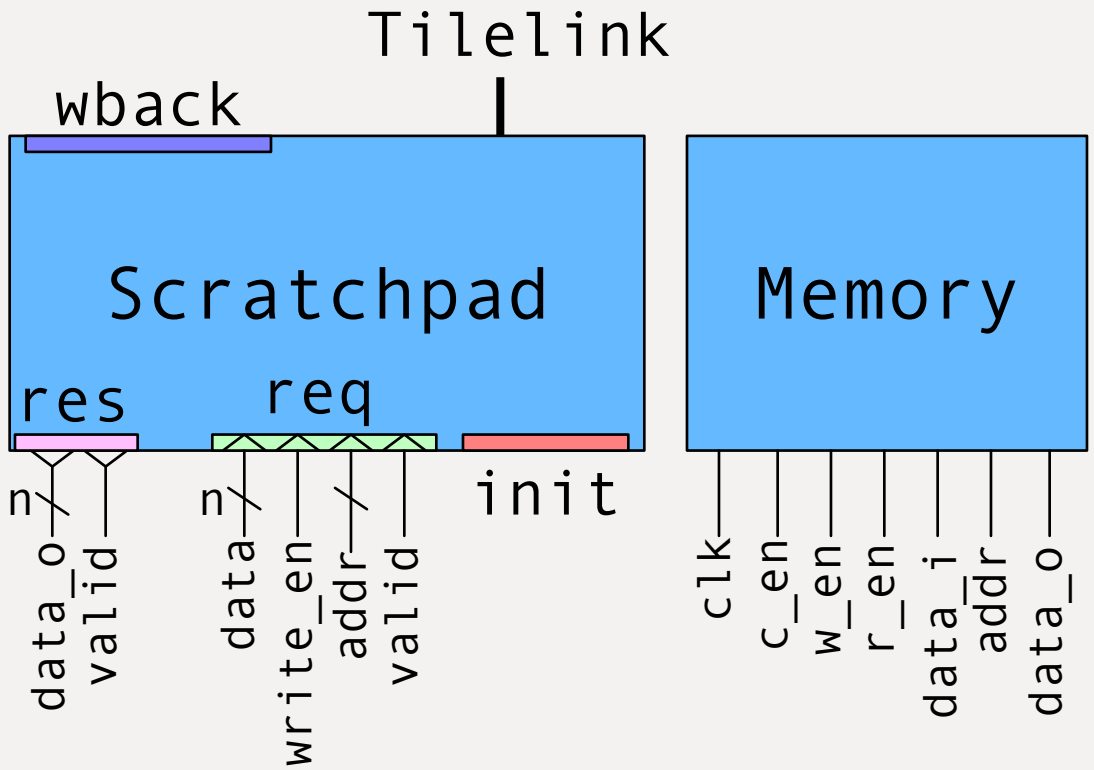

```
address: 0x1030
len    : 0x20
```

Memories - Interfaces



- BRAM vs URAM vs SRAM(ASIC)
  - Single/Dual-Ported memories
  - Read-Write, Read, Write ports
  - Different speeds/capacities
  - ASIC SRAMs: PPA Tradeoffs
- Beethoven Offerings:
  - **Scratchpad memories (Beethoven-backend-managed)**
    - Gives typical SRAM interfaces with additional utilities (init, writeback)

Memories - Interfaces



- BRAM vs URAM vs SRAM(ASIC)
  - Single/Dual-Ported memories
  - Read-Write, Read, Write ports
  - Different speeds/capacities
  - ASIC SRAMs: PPA Tradeoffs
- Beethoven Offerings:
  - Scratchpad memories (Beethoven-backend-managed)
    - Gives typical SRAM interfaces with additional utilities (init, writeback)
  - **Memory (User-managed)**
    - Uses Beethoven's backend to instantiate, but gives user full control to manage
    - Lower-level, SRAM-like interface

Duke

Memories - Configuration

Memory: Inside-RTL

```
val my_mem: MemoryIOBundle = Memory(
    latency = 2,              // n-cycles
    dataWidth = 64,           // width of data bus
    nRows = 1024,             // # of rows
    nReadPorts = 0,           // read-only ports
    nWritePorts = 0,          // write-only ports
    nReadWritePorts = 2,      // RW ports
    withWriteEnable = false,
    debugName: Option[String] = None,
    allowFallbackToRegister: Boolean = true)
```

optional

Scratchpad: Config + Fetch

```
ScratchpadConfig(
  name = “mySPad”,
  dataWidthBits = 64,
  nDatas = 1024,
  nPorts = 2,
  latency = 2,
  features = ScratchpadFeatures())
```

optional

```
val mySpad: ScratchpadModuleChannel =
  getScratchpad (“mySPad”)
```

Memories - Configuration

- Latency - Larger single-cycle memories can be banked/cascaded into larger memories

Memory: Inside-RTL

```
val my_mem: MemoryIOBundle = Memory(
    latency = 2,              // n-cycles
    dataWidth = 64,           // width of data bus
    nRows = 1024,             // # of rows
    nReadPorts = 0,           // read-only ports
    nWritePorts = 0,          // write-only ports
    nReadWritePorts = 2,      // RW ports
    withWriteEnable = false,
    debugName: Option[String] = None,
    allowFallbackToRegister: Boolean = true)
```

optional

Scratchpad: Config + Fetch

```
ScratchpadConfig(
  name = "mySPad",
  dataWidthBits = 64,
  nDatas = 1024,
  nPorts = 2,
  latency = 2,
  features = ScratchpadFeatures())
```

optional

```
val mySpad: ScratchpadModuleChannel =
  getScratchpad ("mySPad")
```

Memories - Configuration

- Latency - Larger single-cycle memories can be banked/cascaded into larger memories
- Data-Width/Columns

Memory: Inside-RTL

```
val my_mem: MemoryIOBundle = Memory(
    latency = 2,              // n-cycles
    dataWidth = 64,           // width of data bus
    nRows = 1024,             // # of rows
    nReadPorts = 0,           // read-only ports
    nWritePorts = 0,          // write-only ports
    nReadWritePorts = 2,      // RW ports
    withWriteEnable = false,
    debugName: Option[String] = None,
    allowFallbackToRegister: Boolean = true)
```

optional

Scratchpad: Config + Fetch

```
ScratchpadConfig(
  name = "mySPad",
  dataWidthBits = 64,
  nDatas = 1024,
  nPorts = 2,
  latency = 2,
  features = ScratchpadFeatures())
```

optional

```
val mySpad: ScratchpadModuleChannel =
  getScratchpad ("mySPad")
```

Memories - Configuration

- Latency - Larger single-cycle memories can be banked/cascaded into larger memories
- Data-Width/Columns
- Memory Depth/Rows

Memory: Inside-RTL

```
val my_mem: MemoryIOBundle = Memory(
    latency = 2,              // n-cycles
    dataWidth = 64,           // width of data bus
    nRows = 1024,             // # of rows
    nReadPorts = 0,           // read-only ports
    nWritePorts = 0,          // write-only ports
    nReadWritePorts = 2,      // RW ports
    withWriteEnable = false,
    debugName: Option[String] = None,
    allowFallbackToRegister: Boolean = true)
```

optional

Scratchpad: Config + Fetch

```
ScratchpadConfig(
  name = "mySPad",
  dataWidthBits = 64,
  nDatas = 1024,
  nPorts = 2,
  latency = 2,
  features = ScratchpadFeatures())
```

optional

```
val mySpad: ScratchpadModuleChannel =
  getScratchpad ("mySPad")
```

Memories - Configuration

- Latency - Larger single-cycle memories can be banked/cascaded into larger memories
- Data-Width/Columns
- Memory Depth/Rows
- Number of ports: R/W/RW

Memory: Inside-RTL

```
val my_mem: MemoryIOBundle = Memory(
    latency = 2,              // n-cycles
    dataWidth = 64,           // width of data bus
    nRows = 1024,             // # of rows
    nReadPorts = 0,           // read-only ports
    nWritePorts = 0,          // write-only ports
    nReadWritePorts = 2,      // RW ports
    withWriteEnable = false,
    debugName: Option[String] = None,
    allowFallbackToRegister: Boolean = true)
```

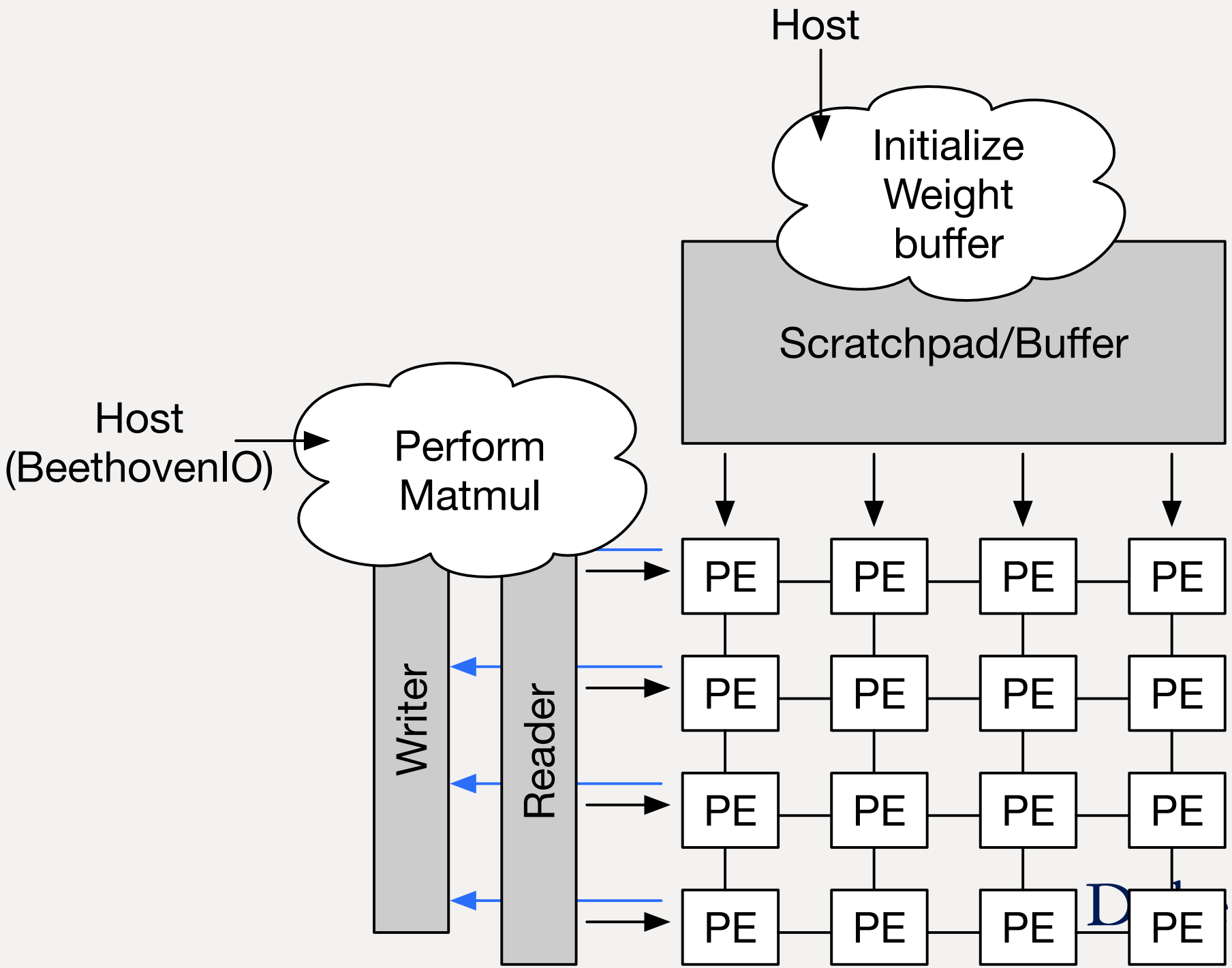
optional

Scratchpad: Config + Fetch

```
ScratchpadConfig(
  name = "mySPad",
  dataWidthBits = 64,
  nDatas = 1024,
  nPorts = 2,
  latency = 2,
  features = ScratchpadFeatures())
```

optional

```
val mySpad: ScratchpadModuleChannel =
  getScratchpad ("mySPad")
```


Host
Initialize
Weight
buffer
Scratchpad/Buffer
Host
(BeethovenIO)
Perform
Matmul
Writer
Reader
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE

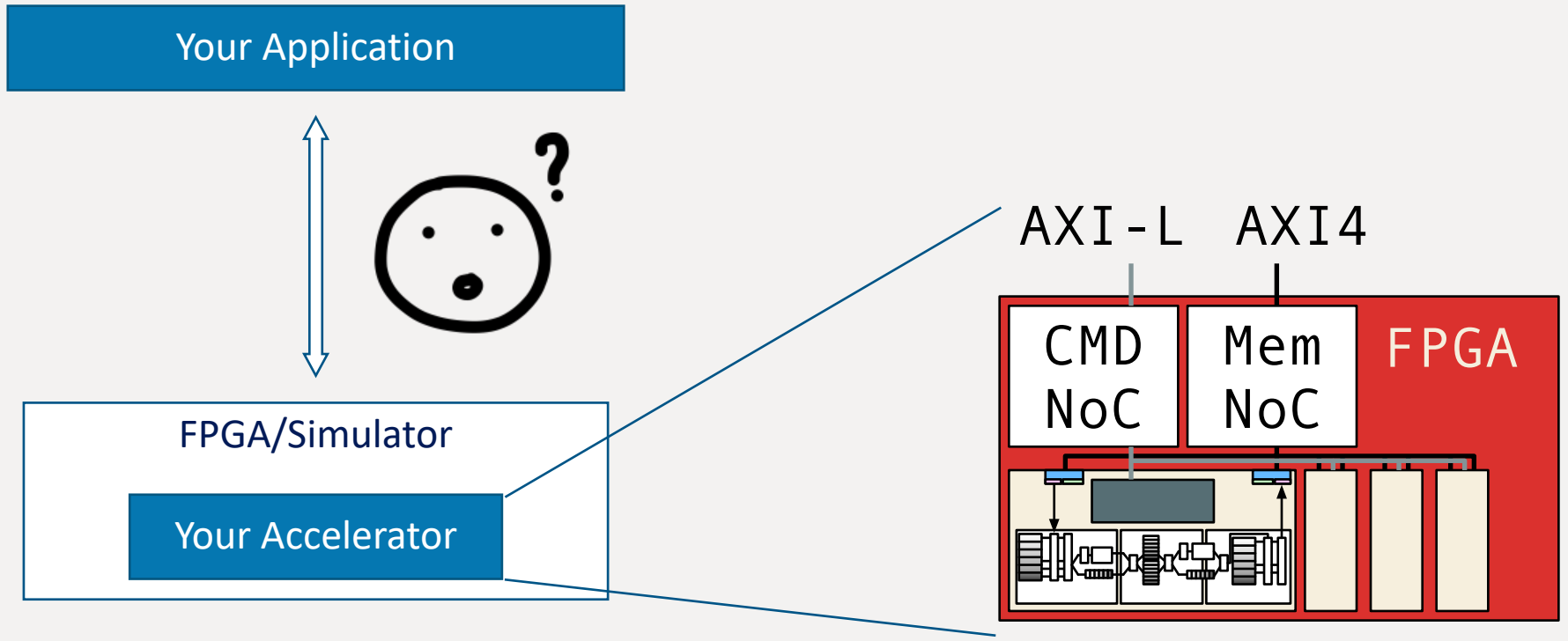
Beethoven

Software Stack and Testing

Duke

Hardware Accelerator Integration

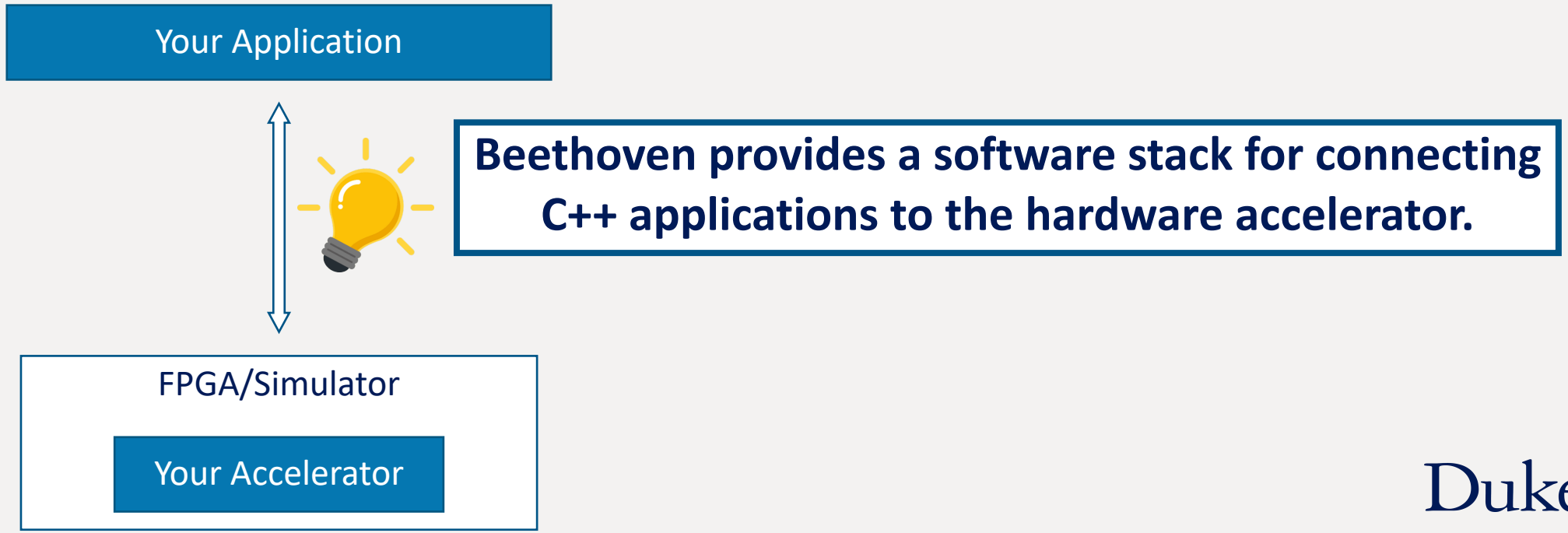
- Beethoven provides us with a ready-to deploy accelerator RTL, but...
- **How do we test/debug this?**
- **How does software interact with our hardware? Or, how to write the software?**



Duke

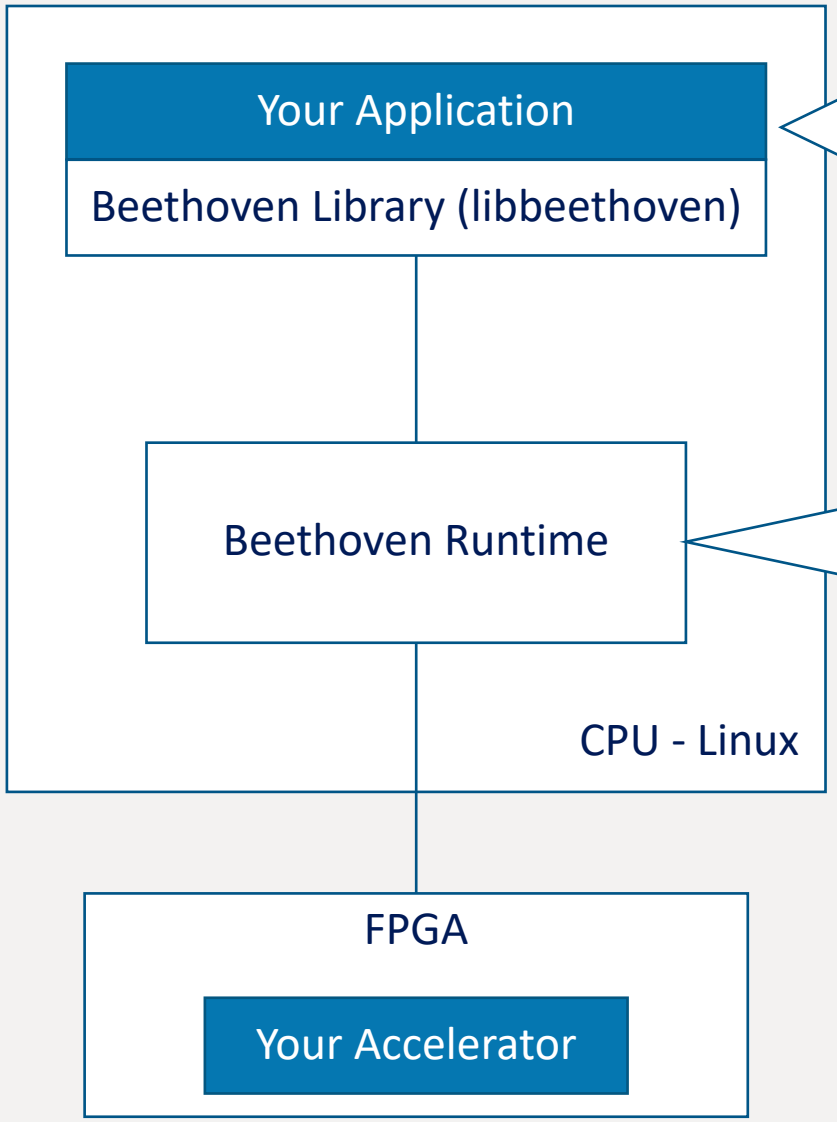
Hardware Accelerator Integration

- Beethoven provides us with a ready-to deploy accelelerator RTL, but...
- **How do we test/debug this?**
- **How does software interact with our hardware? Or, how to write the software?**



Duke

Software Stack Overview



Provides **memory allocation** utilities for accelerator-accessible regions and **routines** for sending Beethoven command/responses

`Beethoven-Software/libbeethoven`

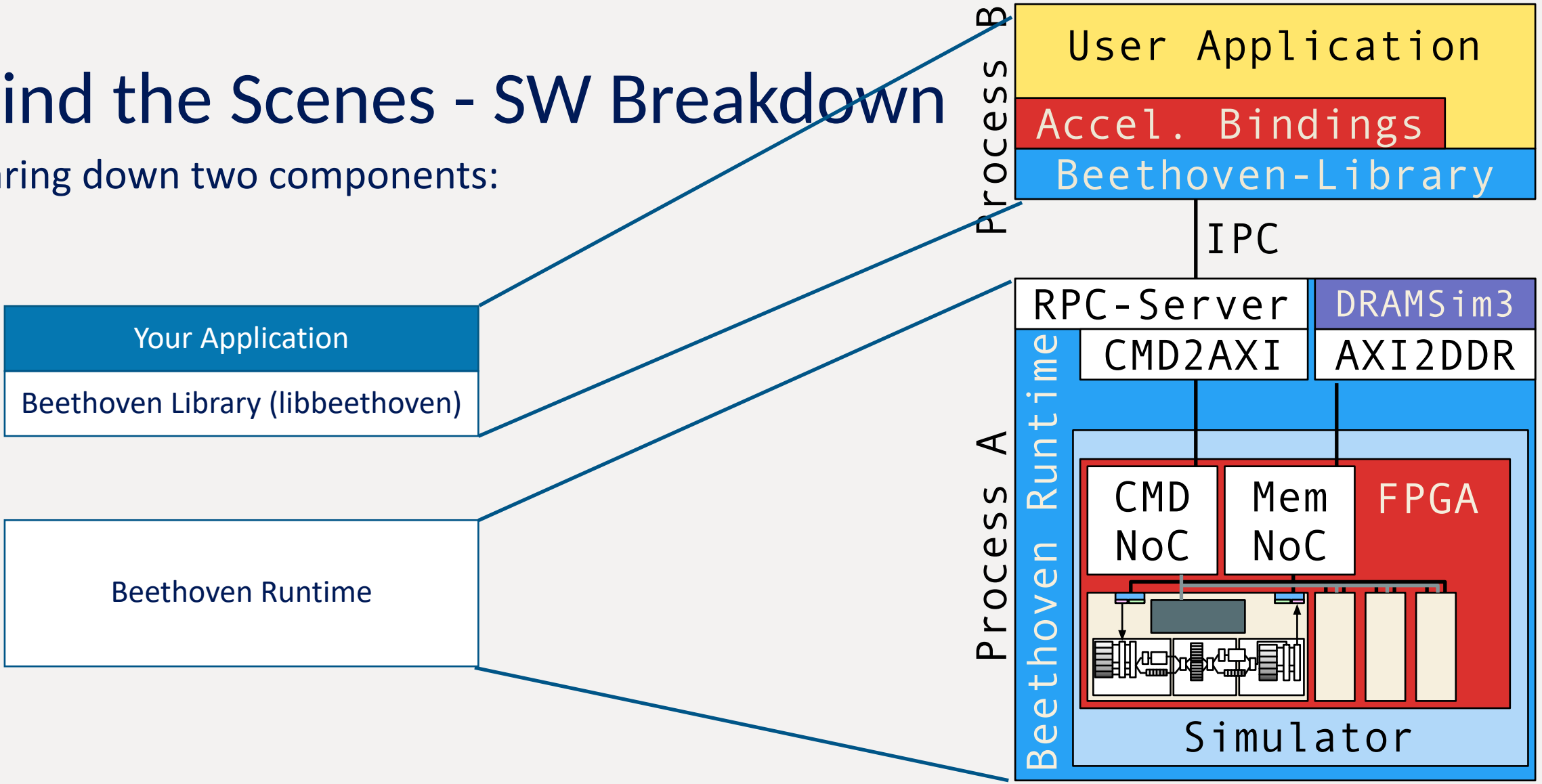
Handles accelerator device **management** and **communication** infrastructure

`Beethoven-Software/runtime`

Duke

Behind the Scenes - SW Breakdown

Tearing down two components:

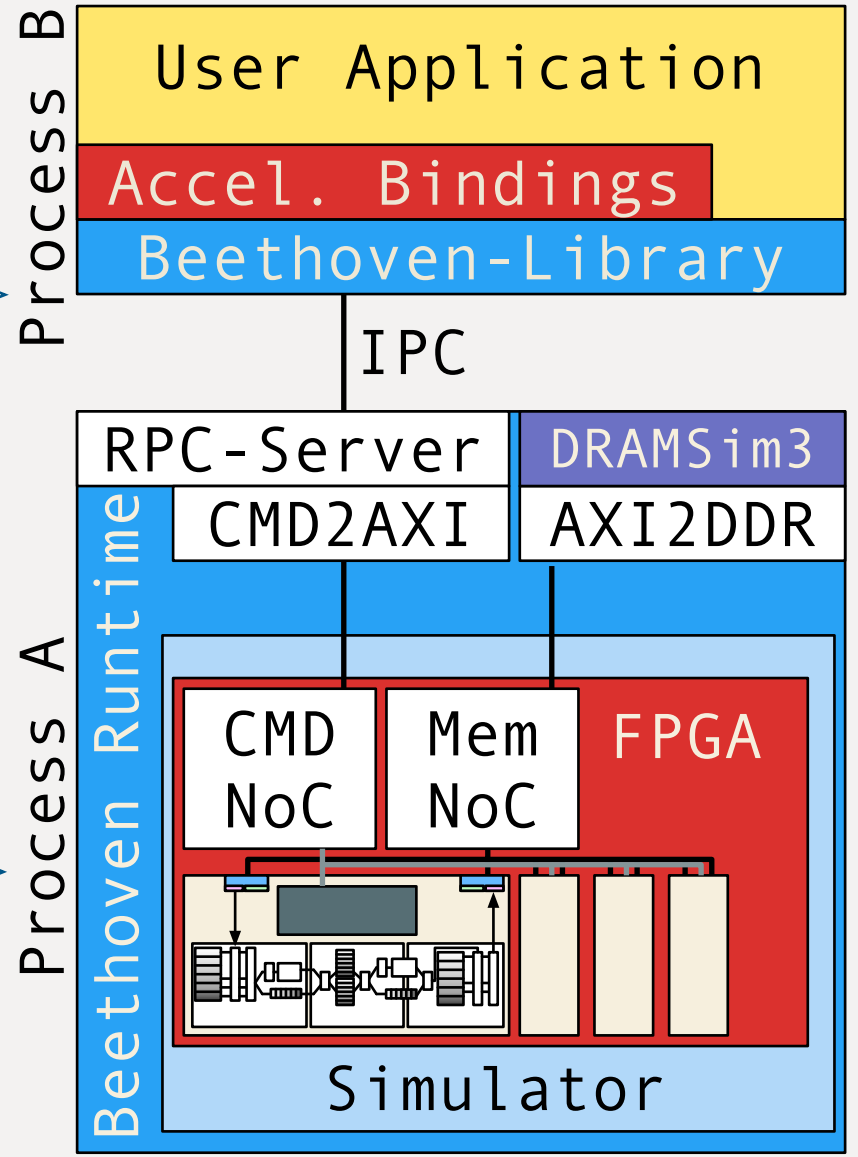


Duke

Behind the Scenes - SW Breakdown

- Beethoven Library (libbeethoven)
  - Userspace library that connects application to runtime
  - Memory allocator for FPGA's memory space

- Beethoven Runtime
  - RPC Server receives commands from user application via inter-process communication (IPC)
  - Controllers for memory and command
  - Simulates RTL using VCS/Icarus
  - Uses DRAMsim3* for cycle-accurate DRAM modeling



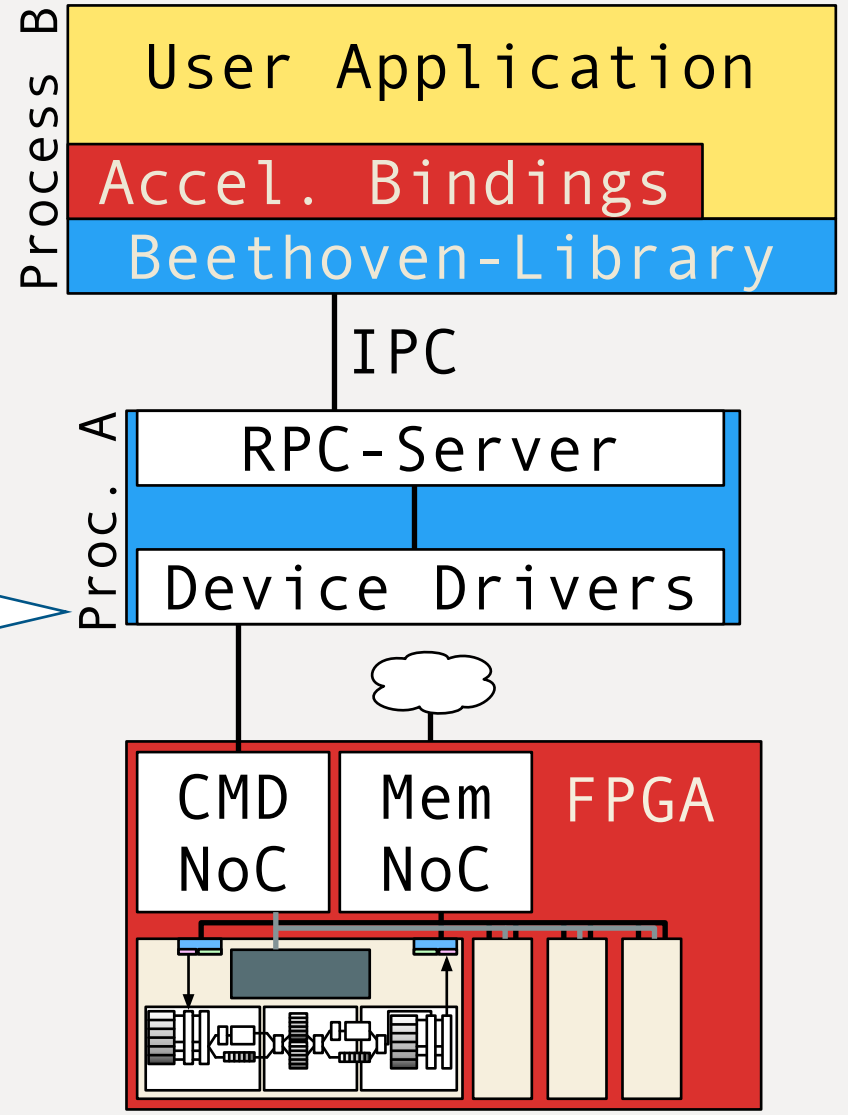
Duke

* Li, S., Yang, Z., Reddy D., Srivastava, A. and Jacob, B., (2020) DRAMsim3: a Cycle-accurate, Thermal-Capable DRAM Simulator, IEEE Computer Architecture Letters.

Behind the Scenes - SW Breakdown

The key differences between simulation and deployment:

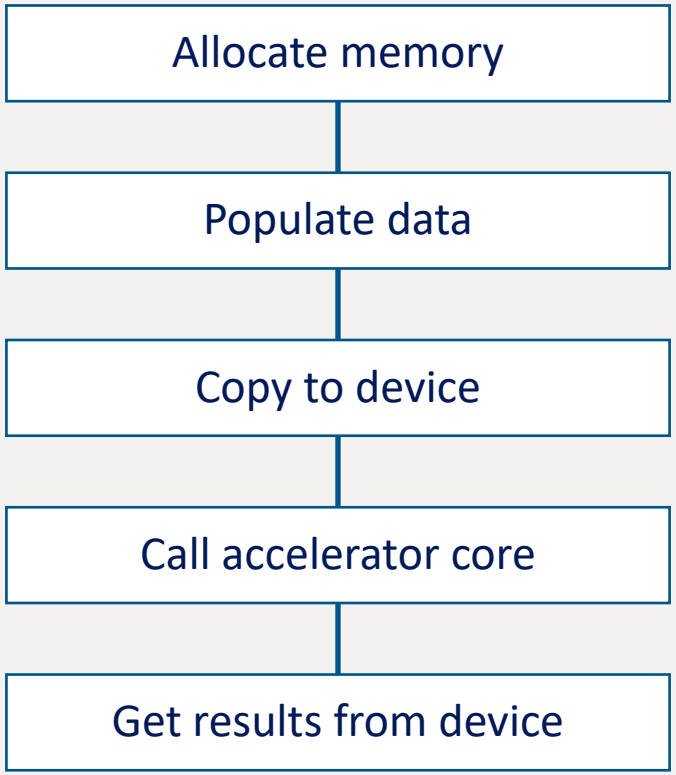
Runtime uses device-specific drivers (PCIE/DMA) to communicate with FPGA instead of using simulator.



Duke

Developing the Control Program/Application

- Minimum example of offloading computation to FPGA



Duke

Developing Control Program

- Stubs generated automatically for userspace calls

Hardware definition

```
class Project1Core()(implicit p: Parameters)
extends AcceleratorCore {
  val my_io = BeethovenIO(new
AccelCommand("project1") {
    val vec_a_addr = Address()
    val vec_b_addr = Address()
    val vec_out_addr = Address()
    val vector_length = UInt(32.W)
  }, EmptyAccelResponse())
```

(hw/Project1Core.scala)

Software definition

```
namespace myProject1 {
beethoven::response_handle<bool> project1(uint16_t core_id,
                                          beethoven::remote_ptr vec_a_addr,
                                          beethoven::remote_ptr vec_b_addr,
                                          beethoven::remote_ptr vec_out_addr,
                                          uint32_t vector_length);

};
```

(target/binding/beethoven_hardware.h)

This is the basic building block for the application that we'll build.

Duke

The Bare Minimum of sw/testbench.cc

- `fpga_handle_t` is the main interface to control and communicate with your FPGA accelerator.
- It provides the c++ building blocks for acceleration:
  - Malloc
  - Data Transfer
  - Kernel Invocation
- Internally launches:
  - Command Channel (`cmd_server`)
  - and Data Channel (`data_server`) using shared memory to minimize data copying overhead
  - (you might see these terms in verbose logs)

```
#include <cstdio>
#include <beethoven/fpga_handle.h>
#include <beethoven_hardware.h>

using namespace beethoven;

int main() {
  fpga_handle_t handle;
  handle.shutdown();
}
```

Duke

Memory Management

```
auto : beethoven::remote_ptrvec_a = handle.malloc(len: size_of_int * n_eles);
auto : beethoven::remote_ptrvec_b = handle.malloc(len: size_of_int * n_eles);
auto : beethoven::remote_ptrvec_out = handle.malloc(len: size_of_int * n_eles);
```

Malloc in host address space

```
auto : int *vec_a_host = (int*)vec_a.getHostAddr();
auto : int *vec_b_host = (int*)vec_b.getHostAddr();
```

Fetch this pointer

```
for (int i = 0; i < n_eles; ++i) {
    vec_a_host[i] = i + 1;
    vec_b_host[i] = i * 2;
}
```

Initialize the data

```
handle.copy_to_fpga(dst: vec_a);
handle.copy_to_fpga(dst: vec_b);
```

Copy to device

File location: your_project/sw/testbench.cc

Duke

Memory Management

Malloc in host address space

```
auto : beethoven::remote_ptrvec_a = handle.malloc(len: size_of_int * n_eles);
auto : beethoven::remote_ptrvec_b = handle.malloc(len: size_of_int * n_eles);
auto : beethoven::remote_ptrvec_out = handle.malloc(len: size_of_int * n_eles);
```

File location: your_project/sw/testbench.cc

Why `remote_ptr`? - Ensures seamless host <> accelerator memory use

- A **smart pointer** with shared_ptr semantics - **No manual free() needed**
- Abstracts away memory access differences:
  - **Shared memory systems** (e.g. Zynq)
  - **Discrete memory systems** (e.g. AWS F2)

Duke

Memory Management

```
auto : int *vec_a_host = (int*)vec_a.getHostAddr();
auto : int *vec_b_host = (int*)vec_b.getHostAddr();
for (int i = 0; i < n_eles; ++i) {
    vec_a_host[i] = i + 1;
    vec_b_host[i] = i * 2;
}
handle.copy_to_fpga(dst: vec_a);
```

Fetch the pointers

Initialize the data

`.getHostAddr` gives a regular host pointer for initialization

Tips for building efficient accelerator:

Batching more data in one transaction is usually more efficient than having multiple smaller transactions. Batching is desired because it amortizes the latency of control signals.

File location: your_project/sw/testbench.cc

Duke

Launch Accelerator Core with One Line!

```
myProject1::project1(0, vec_a, vec_b, vec_out, n_eles);
```

File location: your_project/sw/testbench.cc

Duke

Launch Accelerator Core

```
myProject1::project1(0, vec_a, vec_b, vec_out, n_eles);
```

`core_id` selects which core (if multi-core accelerator) executes the task.

```
auto : beethoven::re..._handle<bool>resp_handle = myVectorAdd::vector_add(core_id: 0,
                                                  vec_a_addr: vec_a,
                                                  vec_b_addr: vec_b,
                                                  vec_out_addr: vec_out,
                                                  vector_length: n_eles);
```

File location: your_project/sw/testbench.cc

Launches asynchronously - does not block the host!
`resp_handle` is a future-like handle for the accelerator task

Duke

Fetching Results

.get blocks until completion

⚠️ If your accelerator doesn't return a response, get() will throw an error

```
auto : boolresponse = resp_handle.get();
handle.copy_from_fpga(src: vec_out);
```

Tip: Non-blocking Check with try_get:

```
auto maybe_response = resp_handle.try_get();
// Returns std::optional
```

File location: your_project/sw/testbench.cc

Duke

Advanced: Library Integration

- The `beethoven_build` function makes an executable, but what if you want to build a library (e.g. for Python usage)?
- In this case, `link_beethoven_to_target` is what we need.
- Here's a minimum CMakeLists.txt file.

```
cmake_minimum_required(VERSION 3.15)
project(beethoven_python)

find_package(beethoven REQUIRED)
find_package(pybind11 REQUIRED)

set(CMAKE_CXX_STANDARD 17)

# Python binding module
pybind11_add_module(beethoven_python
    python_bindings.cpp
)

link_beethoven_to_target(beethoven_python)
```

project1/sw/CMakeLists.txt

Duke

Try your systolic array example

```
$ beethoven sim systolic_array_tb

start servers
finished init structures
     Running systolic_array_tb
[exec] BEETHOVEN_PROJECT_ROOT=/home/mason/Developer/Projects/Beethoven-
refurbish/Beethoven-Zoo/SystolicArray-Verilog
/home/mason/Developer/Projects/Beethoven-refurbish/Beethoven-Zoo/SystolicArray-
Verilog/target/sw/systolic_array_tb
enqueueing command: 0
enqueueing command: 1
enqueueing command: 2
[PASS] systolic_array: 8 x 8 matmul with inner dimension 16 matched across weight
reuse.
    Stopping BeethovenRuntime
✓ done.
```

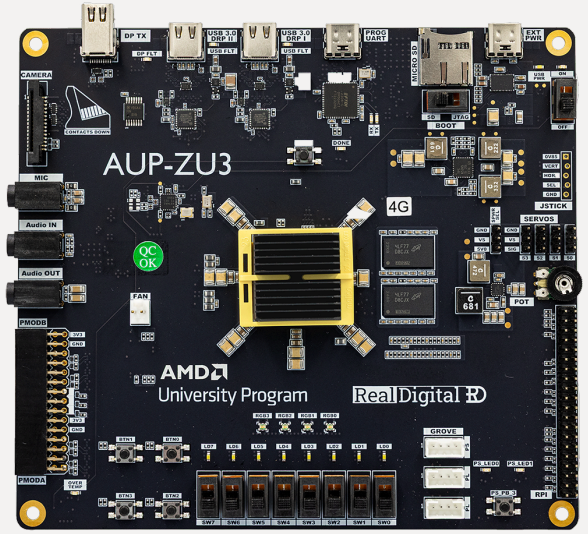
Duke

Beethoven

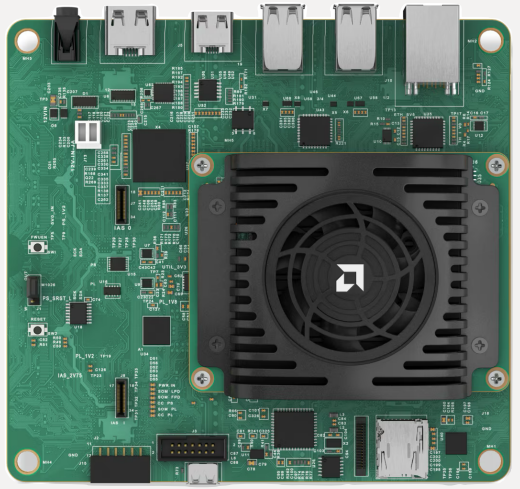
Running your project on Embedded and Datacenter FPGA

Duke

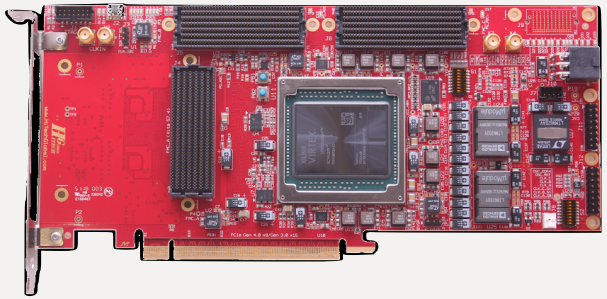
From Edge to Datacenter



AUP-ZU3, the cheapest Ultrascale+ Zynq board
4x Cortex A53 & Programming Logic



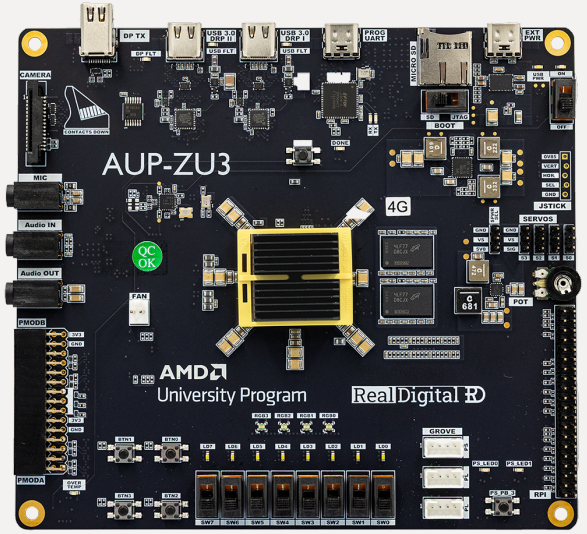
Kria K26, the affordable mid-tier Zynq



AWS-F2 hosts datacenter-grade FPGA (Virtex UltraScale+ VU47P)
Discrete card

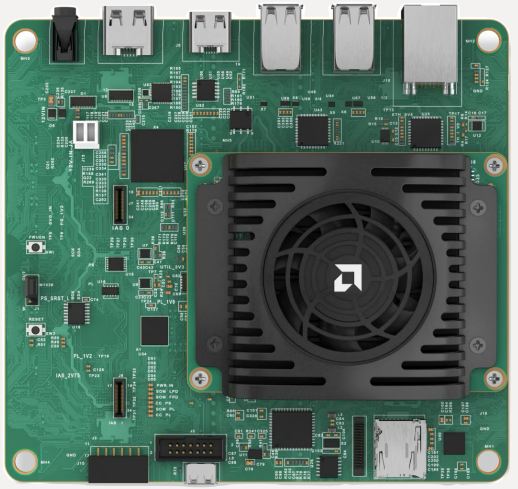
Duke

From Edge to Datacenter

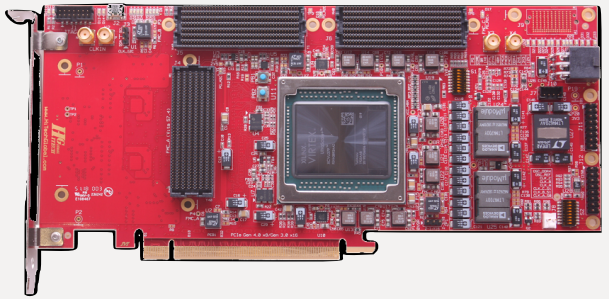


AUP-ZU3, the cheapest Ultrascale+ Zynq board
4x Cortex A53 & Programming Logic

Experience today!
Short synthesis time



Kria K26, the affordable mid-tier Zynq



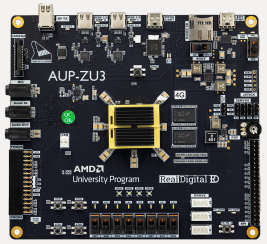
AWS-F2 hosts datacenter-grade FPGA
(Virtex UltraScale+ VU47P)
Discrete card

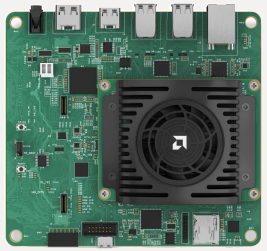
Experience today! It's on us.
Recommended for advanced users

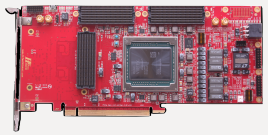
Have your own platform?

All platform-specific files located at:

Beethoven-Hardware/src/main/scala/beethoven/Platforms/FPGA







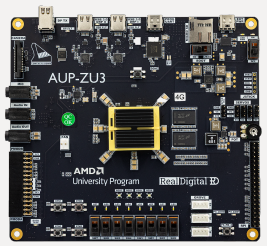
```
$ pwd
Beethoven-Hardware/src/main/scala/beethoven/Platforms/FPGA/Xilinx
$ tree -L 2
.
├── AWS
│   ├── AWSF1Platform.scala
│   ├── AWSF2Platform.scala
│   ├── DMAHelper.scala
│   ├── MemsetHelper.scala
│   ├── Shell.scala
│   └── SynthScript.scala
├── U200Platform.scala
├── Zynq
│   ├── AUPZU3Platform.scala
│   └── KriaPlatform.scala
└── package.scala
```

Duke

Have your own platform?

All platform-specific files located at:

Beethoven-Hardware/src/main/scala/beethoven/Platforms/FPGA

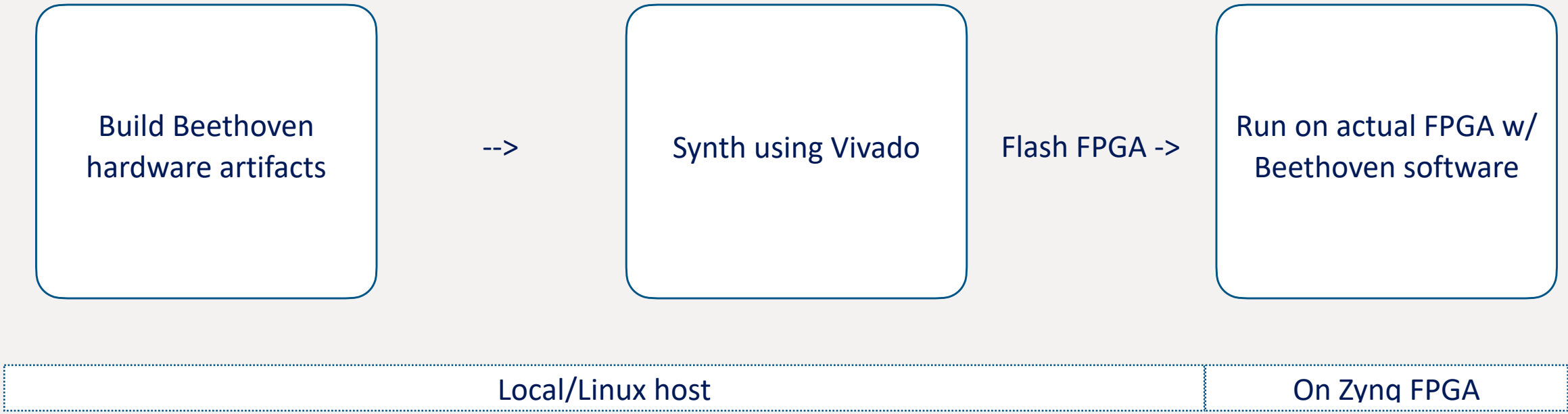



```
override val platformType: PlatformType = PlatformType.FPGA
override val hasDiscreteMemory: Boolean = false
override val isActiveHighReset: Boolean = false

override val frontBusBaseAddress: Long = 0x2000000000L
override val frontBusAddressNBits: Int = 40
override val frontBusAddressMask: Long = 0xffffL
override val frontBusBeatBytes: Int = 16
override val frontBusProtocol: FrontBusProtocol = new AXIFrontBusProtocol
```

Example platform definition for ZU3

Duke

ZYNQ - Workflow Overview



Beethoven CLI tool has built-in commands to go through these steps!

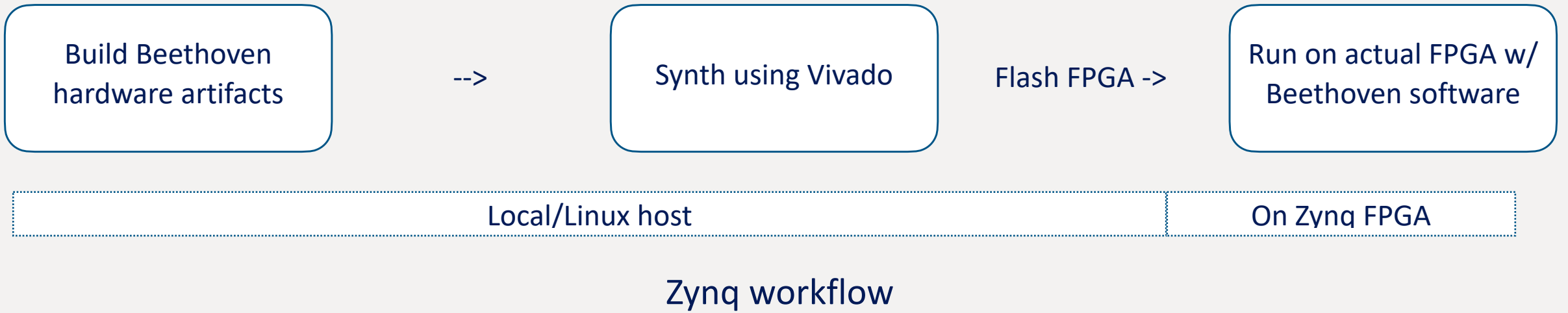
Duke

ZYNQ - Workflow

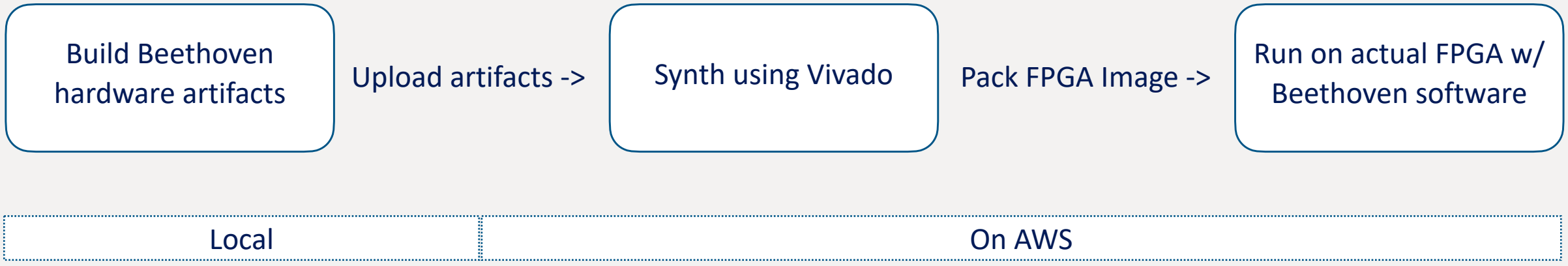
```
# On Linux with Vivado:
$ beethoven build hw --release        # build hardware artifacts
$ source /opt/Xilinx/2025.1/Vivado/settings64.sh # Setup Vivado
$ beethoven synth                                # Auto synth
$ beethoven flash                                # Flash to FPGA

# On FPGA:
$ sudo beethoven run                # Start testbench and runtime
```

Duke

Comparison w/ F2 Workflow



Zynq workflow



F2 workflow

F2 - Workflow (simplified)

```
# Locally:
$ beethoven build hw --release                          # build hardware artifacts
$ beethoven aws upload --host 10.0.0.2 --key ./ssh-key.pem # Upload artifacts to AWS

# On AWS with Vivado:
$ beethoven setup                                       # Setup Beethoven
$ source /opt/Xilinx/2025.1/Vivado/settings64.sh        # Setup Vivado
$ cd ~/cl_beethoven_top/build/scripts
$ python3 aws_build_dcp_from_cl.py --cl cl_beethoven_top # Launch Vivado and synthesis
$ beethoven aws create-fpga-image --name my-project     # Create FPGA image

# On AWS F2 Instance:
$ beethoven aws load --agfi agfi-01337c0515461e0f2      # Load FPGA image
$ sudo beethoven run                                    # Start testbench and runtime
```

Duke

Live demo: ZYNQ deployment

Duke

Vivado Box's IP address: 172.22.211.48

Use:
ssh username@172.22.211.48
to access it.
Synth commands are on cheatsheet.

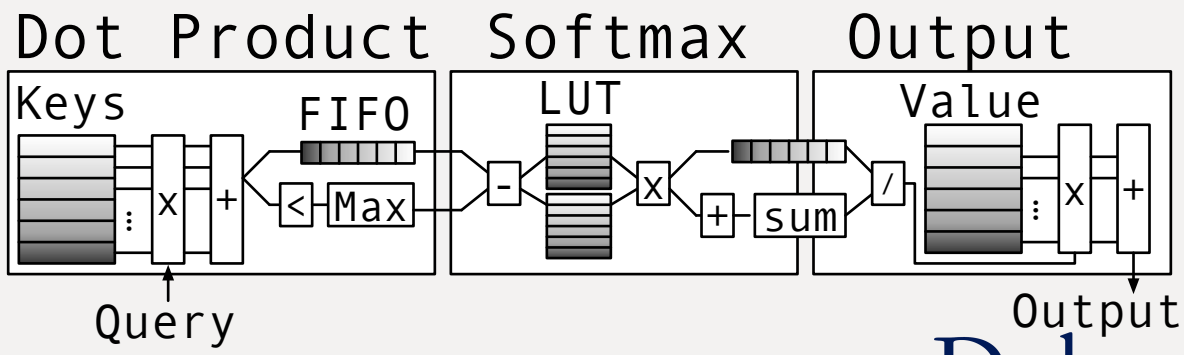
Accelerator System Overview

Code Structures

- Typically, we structure hardware designs as **modules**
  - Separation of control from compute pipeline
  - Separation of IPs from core logic

```
module DotProduct(
  input clk,
  input rst,
  input [15:0] query,
  output [15:0] max_value,
  ...)
```

Example - A³ attention accelerator pipeline

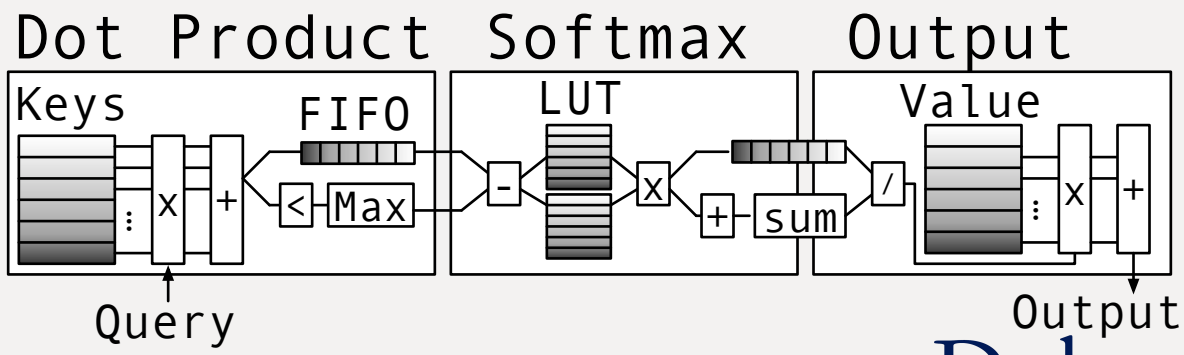


Code Structures

- Typically, we structure hardware designs as **modules**
  - Separation of control from compute pipeline
  - Separation of IPs from core logic
- Modular design aligns well with pipelined logic

```
module DotProduct(
  input clk,
  input rst,
  input [15:0] query,
  output [15:0] max_value,
  ...)
```

Example - A³ attention accelerator pipeline



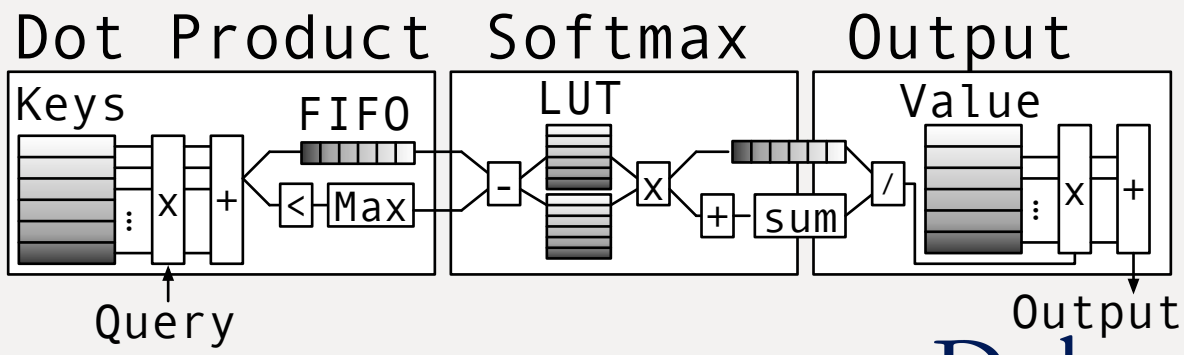
Duke

Code Structures

- Typically, we structure hardware designs as **modules**
  - Separation of control from compute pipeline
  - Separation of IPs from core logic
- Modular design aligns well with pipelined logic
  - *This pipeline is usually the core focus of your work*

```
module DotProduct(
  input clk,
  input rst,
  input [15:0] query,
  output [15:0] max_value,
  ...)
```

Example - A³ attention accelerator pipeline

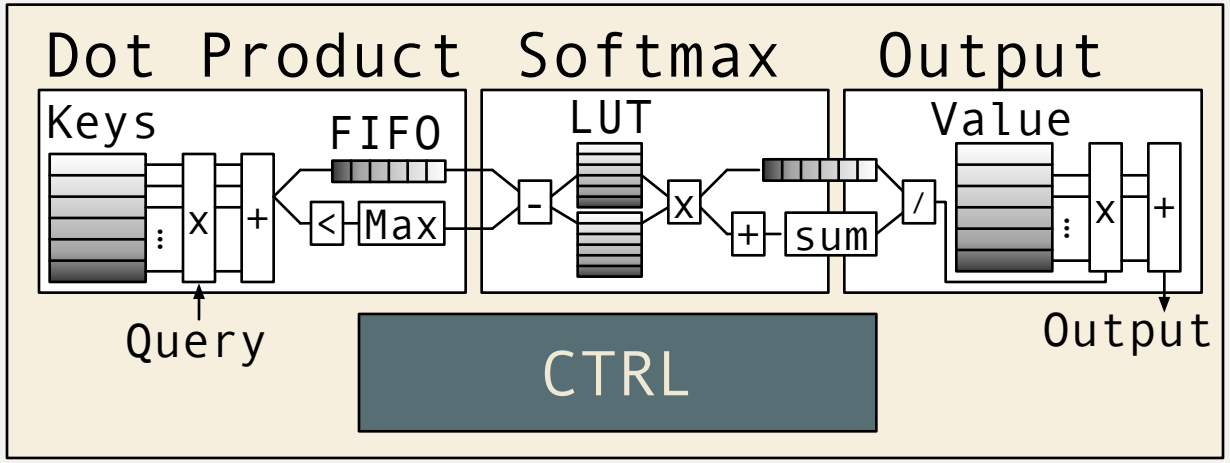


Duke

Code Structures - Accelerator Cores

- An **Accelerator Core** is the top-level block that contains application-specific control/logic
  - Instantiates the compute pipeline
  - Manages communication between host/memory and the compute pipeline

A³ attention accelerator core

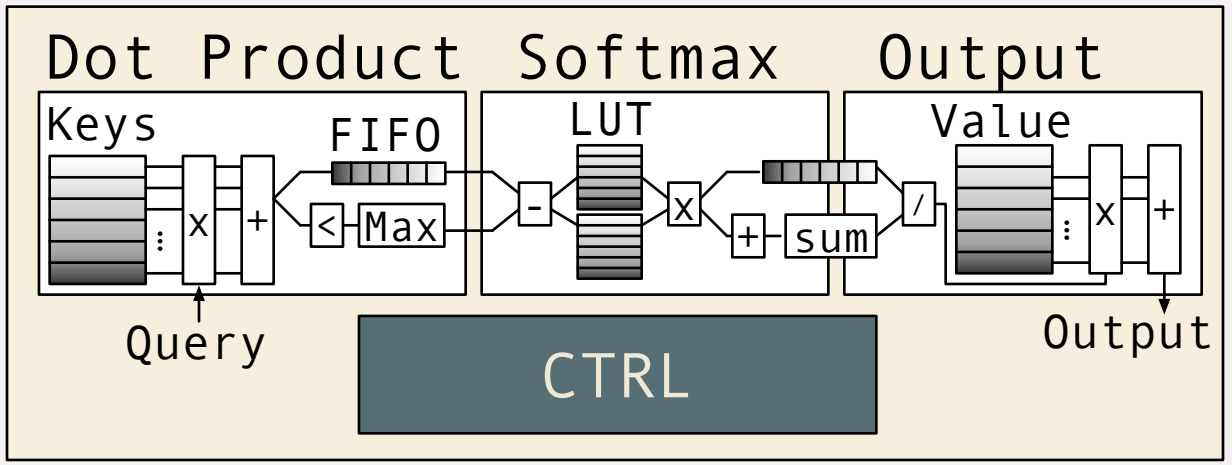


Duke

Code Structures - Accelerator Cores

- An **Accelerator Core** is the top-level block that contains application-specific control/logic
  - Instantiates the compute pipeline
  - Manages communication between host/memory and the compute pipeline
- Beethoven formalizes the accelerator core abstraction, providing...

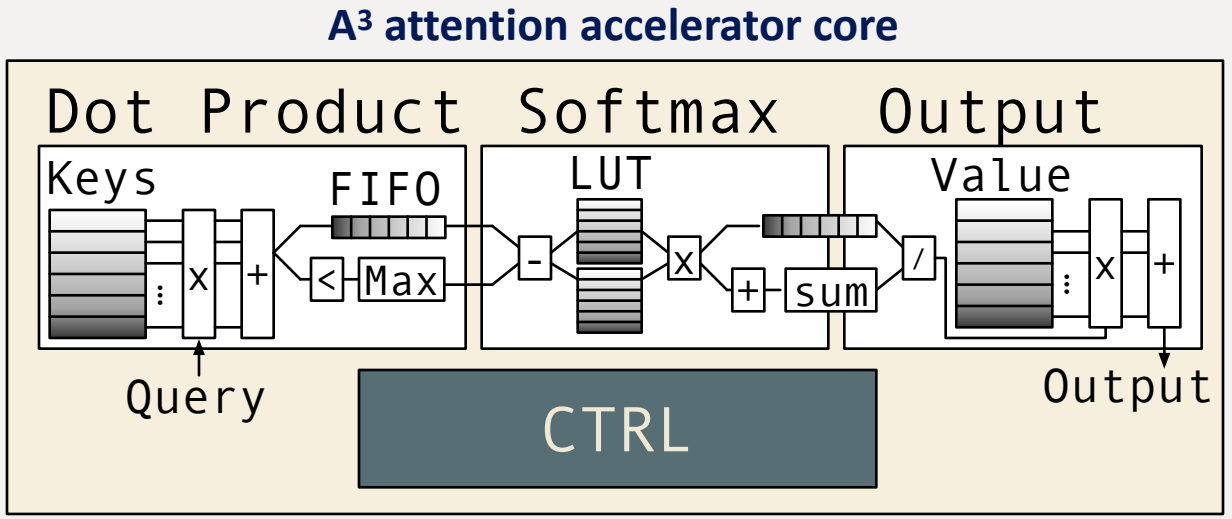
A³ attention accelerator core



Duke

Code Structures - Accelerator Cores

- An **Accelerator Core** is the top-level block that contains application-specific control/logic
  - Instantiates the compute pipeline
  - Manages communication between host/memory and the compute pipeline
- Beethoven formalizes the accelerator core abstraction, providing...
  - RTL abstractions for Host/External Memory communication + On-Chip Memory



Duke

Code Structures - Accelerator Cores

- An **Accelerator Core** is the top-level block that contains application-specific control/logic
  - Instantiates the compute pipeline
  - Manages communication between host/memory and the compute pipeline
- Beethoven formalizes the accelerator core abstraction, providing...
  - RTL abstractions for Host/External Memory communication + On-Chip Memory
  - Software abstractions for directly interacting with accelerator cores

A³ attention accelerator core