PowerPULP Hands-on Session

OPRECOMP at NiPS Summer School

Fabian Schuiki, Stefan Mach
ETH Zürich

July 16-20, 2018 in Perugia, Italy

This project is co-funded by the European Union's H2020-EU.1.2.2. - FET Proactive research and innovation programme under grant agreement #732631.

Virtual Machine

- We will do some tinkering during the next 3 hours. So...

Virtual Machine

- We will do some tinkering during the next 3 hours. So...
- Install VirtualBox:

  `https://www.virtualbox.org`

Virtual Machine

- We will do some tinkering during the next 3 hours. So...
- Install VirtualBox:

  `https://www.virtualbox.org`

- Download the following Virtual Machine (one is enough):

  `https://iis-people.ee.ethz.ch/~fschuiki/perugia2018/vm_image.tar.gz`
  `https://iis-people.ee.ethz.ch/~fschuiki/perugia2018/vm_image.zip`

  Username and password: `oprecomp`

Virtual Machine

- We will do some tinkering during the next 3 hours. So...
- Install VirtualBox:

  `https://www.virtualbox.org`

- Download the following Virtual Machine (one is enough):

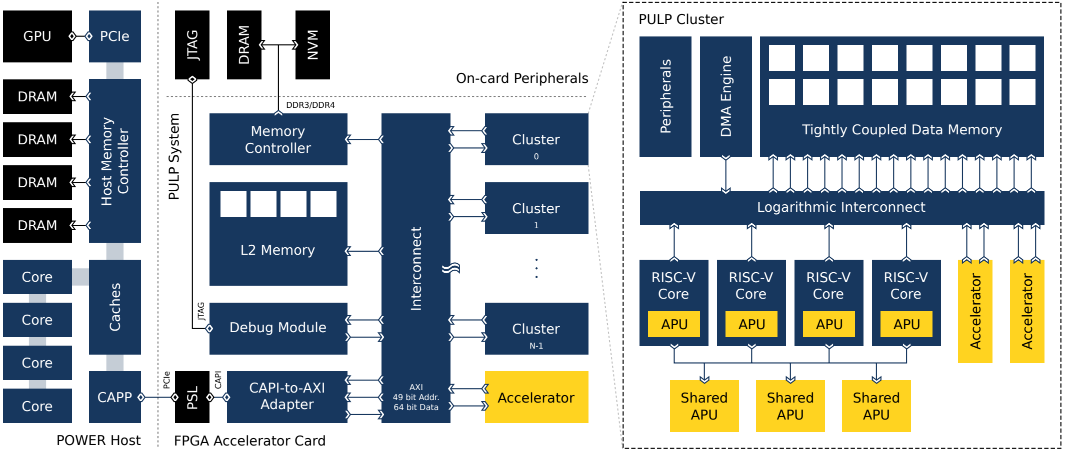
  `https://iis-people.ee.ethz.ch/~fschuiki/perugia2018/vm_image.tar.gz`
  `https://iis-people.ee.ethz.ch/~fschuiki/perugia2018/vm_image.zip`

  Username and password: `oprecomp`

- Schedule:
  - 09:00-10:30 – PowerPULP (aka kW platform, *Fabian Schuiki, ETH*)
  - 11:00-12:30 – GAP (aka mW platform, *Francesco Paci, GWT*)

Introduction

The Big Picture



- Transprecision float unit (float8, float16, float16alt)
- NTX streaming processor (float32 now, others later)
- Dedicated accelerators?

Hardware

Server

IBM POWER8 Minsky:

- Set up with Ubuntu 16.04.2 LTS
- 8K5 installed and tested

Hardware

Server

IBM POWER8 Minsky:

- Set up with Ubuntu 16.04.2 LTS
- 8K5 installed and tested

ETH: no GPU, 8K5 card

IBM: GPU, few 8K5 and many KU3 cards

QUB: GPU, 8K5 card

Hardware

Server

IBM POWER8 Minsky:

- Set up with Ubuntu 16.04.2 LTS
- 8K5 installed and tested

ETH: no GPU, 8K5 card

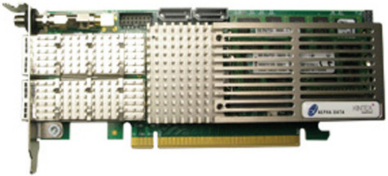
IBM: GPU, few 8K5 and many KU3 cards

QUB: GPU, 8K5 card

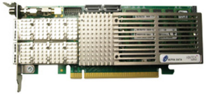
Accelerator Cards

Alpha Data 8K5:

Alpha Data KU3:



Accelerator Cards

		
	8KU	**KU3**
FPGA:	XCKU115-2-FLVA1517E	XCKU060-FFVA1156
CLBs	1451 k	726 k
DSP Slices	5520	2760
Block RAM	75.9 Mbit	38.0 Mbit
DRAM:	16 GiB DDR4-2400	8 GiB DDR3-1600
PCIe:	Gen3 x8	Gen3 x8
PULP Clusters:	4	2
Speed:	50 MHz	50 MHz
Where:	ETH/QUB	IBM Cloud

CAPI

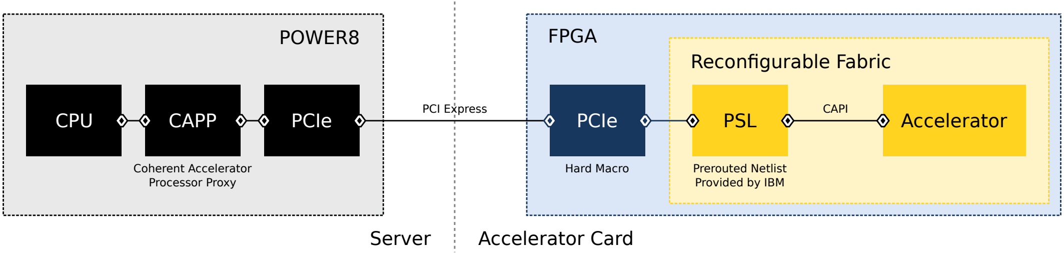
- Coherent Accelerator Processor Interface
- Abstraction for communication between user space and FPGA fabric

CAPI

- Coherent Accelerator Processor Interface
- Abstraction for communication between user space and FPGA fabric
- **libcxl:** API exposed to the user space program
- **CAPI:** Interface exposed to the FPGA fabric

CAPI

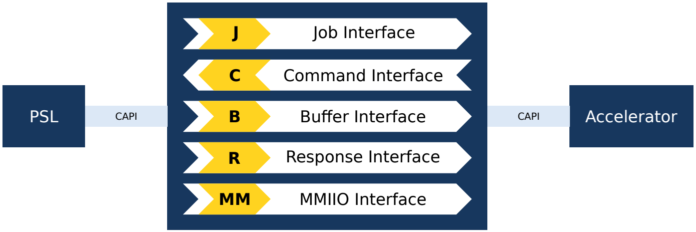
- Coherent Accelerator Processor Interface
- Abstraction for communication between user space and FPGA fabric
- **libcxl:** API exposed to the user space program
- **CAPI:** Interface exposed to the FPGA fabric



CAPI Channels

CAPI consists of five communication channels:

J: Job interface (reset, work element descriptor)

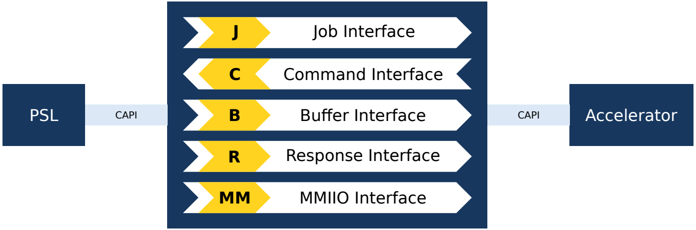


CAPI Channels

CAPI consists of five communication channels:

J: Job interface (reset, work element descriptor)

C: Command interface (read/write requests, cache control, interrupts)



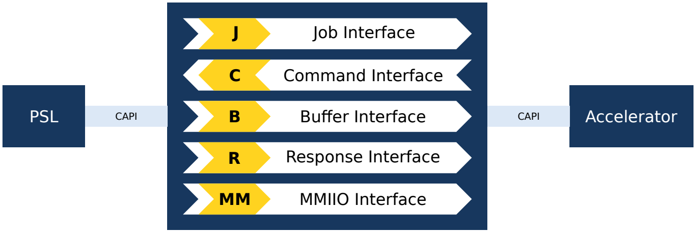
CAPI Channels

CAPI consists of five communication channels:

J: Job interface (reset, work element descriptor)

C: Command interface (read/write requests, cache control, interrupts)

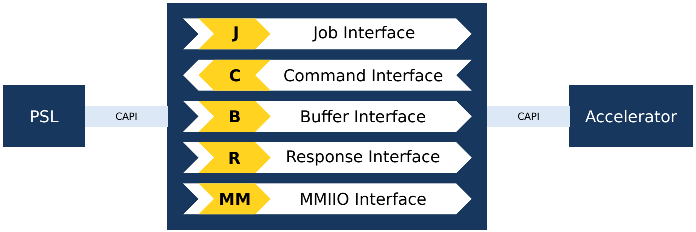
B: Buffer interface (read/write data)



CAPI Channels

CAPI consists of five communication channels:

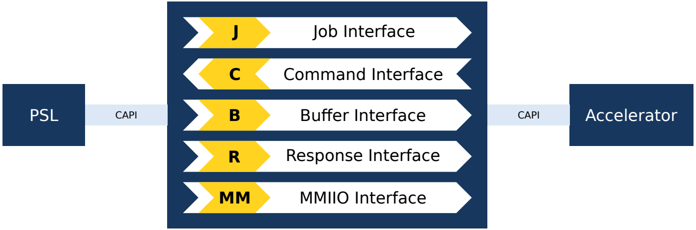
- J: Job interface (reset, work element descriptor)
- C: Command interface (read/write requests, cache control, interrupts)
- B: Buffer interface (read/write data)
- R: Response interface (complementary to the command)



CAPI Channels

CAPI consists of five communication channels:

- J: Job interface (reset, work element descriptor)
- C: Command interface (read/write requests, cache control, interrupts)
- B: Buffer interface (read/write data)
- R: Response interface (complementary to the command)
- MM: MMIO interface (side channel for configuration and register reading/writing)



Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

- Sysfs directory with card information:

```
# ls /sys/class/cxl
afu0.0d

# ls /sys/class/cxl/afu0.0d
afu0.0d/  api_version
  api_version_compatible  cr0/
  device  irqs_max  irqs_min
  mmio_size  mode  modes_supported
  power/  prefault_mode  reset
  subsystem  uevent
```

Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

- Sysfs directory with card information:

```
# ls /sys/class/cxl
afu0.0d

# ls /sys/class/cxl/afu0.0d
afu0.0d/  api_version
  api_version_compatible  cr0/
  device  irqs_max  irqs_min
  mmio_size  mode  modes_supported
  power/  prefault_mode  reset
  subsystem  uevent
```

- AFU Descriptor information:

```
# ls /sys/class/cxl/afu0.0d/cr0
class  config  device  vendor
```

Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

- Sysfs directory with card information:

```
# ls /sys/class/cxl
afu0.0d

# ls /sys/class/cxl/afu0.0d
afu0.0d/  api_version
  api_version_compatible  cr0/
  device  irqs_max  irqs_min
  mmio_size  mode  modes_supported
  power/  prefault_mode  reset
  subsystem  uevent
```

- AFU Descriptor information:

```
# ls /sys/class/cxl/afu0.0d/cr0
class  config  device  vendor
```

- Resetting a card (as root):

```
echo 1 > /sys/class/cxl/afu0.0d/reset
```

Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

- Sysfs directory with card information:

```
# ls /sys/class/cxl
afu0.0d

# ls /sys/class/cxl/afu0.0d
afu0.0d/  api_version
  api_version_compatible  cr0/
  device  irqs_max  irqs_min
  mmio_size  mode  modes_supported
  power/  prefault_mode  reset
  subsystem  uevent
```

- AFU Descriptor information:

```
# ls /sys/class/cxl/afu0.0d/cr0
class  config  device  vendor
```

- Resetting a card (as root):

```
echo 1 > /sys/class/cxl/afu0.0d/reset
```

- Rescan the PCI bus for debugging card connectivity (as root):

```
echo 1 > /sys/bus/pci/rescan
```

Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

- Sysfs directory with card information:

```
# ls /sys/class/cxl
afu0.0d

# ls /sys/class/cxl/afu0.0d
afu0.0d/  api_version
  api_version_compatible  cr0/
  device  irqs_max  irqs_min
  mmio_size  mode  modes_supported
  power/  prefault_mode  reset
  subsystem  uevent
```

- AFU Descriptor information:

```
# ls /sys/class/cxl/afu0.0d/cr0
class  config  device  vendor
```

- Resetting a card (as root):

```
echo 1 > /sys/class/cxl/afu0.0d/reset
```

- Rescan the PCI bus for debugging card connectivity (as root):

```
echo 1 > /sys/bus/pci/rescan
```

- Flash the card (as root):

```
capi-flash-script ~/my_bitstream.bin
```

Interaction with CAPI Accelerator on Linux

- Cards visible as devices:

```
# ls /dev/cxl
afu0.0d
```

- Sysfs directory with card information:

```
# ls /sys/class/cxl
afu0.0d

# ls /sys/class/cxl/afu0.0d
afu0.0d/  api_version
  api_version_compatible  cr0/
  device  irqs_max  irqs_min
  mmio_size  mode  modes_supported
  power/  prefault_mode  reset
  subsystem  uevent
```

- AFU Descriptor information:

```
# ls /sys/class/cxl/afu0.0d/cr0
class  config  device  vendor
```

- Resetting a card (as root):

```
echo 1 > /sys/class/cxl/afu0.0d/reset
```

- Rescan the PCI bus for debugging card connectivity (as root):

```
echo 1 > /sys/bus/pci/rescan
```

- Flash the card (as root):

```
capi-flash-script ~/my_bitstream.bin
```

Caution: Flashing requires working PSL on the FPGA; if a broken image is flashed, the card bricks → JTAG cable required to unbrick.

CXL API

- CAPI accelerators are exposed via the Linux kernel's **cxl** [1] mechanism.

[1] `https://github.com/torvalds/linux/blob/master/include/misc/cxl.h`

CXL API

- CAPI accelerators are exposed via the Linux kernel's **cxl** [1] mechanism.
- IBM provides **libcxl** [2], a convenient user space wrapper library.

```c
// libcxl usage example
#include <libcxl.h>

// Prepare a work element descriptor.
struct {
  float *values;
  size_t num_values;
  uint64_t done;
} wed = { /* ... */ };
wed.done = 0;

// Offload work to accelerator.
struct cxl_afu_h *afu =
  cxl_afu_open_dev("/dev/cxl/afu0.0d");
cxl_afu_attach(afu, (uint64_t)&wed);
while (!wed.done); // better use
  interrupt
cxl_afu_free(afu);
```

[1] https://github.com/torvalds/linux/blob/master/include/misc/cxl.h
[2] https://github.com/ibm-capi/libcxl

CXL API

- CAPI accelerators are exposed via the Linux kernel's **cxl** [1] mechanism.
- IBM provides **libcxl** [2], a convenient user space wrapper library.
- Accelerator must signal completion by writing to one of the WED fields (and possibly raising an interrupt).

```
// libcxl usage example
#include <libcxl.h>

// Prepare a work element descriptor.
struct {
  float *values;
  size_t num_values;
  uint64_t done;
} wed = { /* ... */ };
wed.done = 0;

// Offload work to accelerator.
struct cxl_afu_h *afu =
  cxl_afu_open_dev("/dev/cxl/afu0.0d");
cxl_afu_attach(afu, (uint64_t)&wed);
while (!wed.done); // better use
  interrupt
cxl_afu_free(afu);
```

[1] https://github.com/torvalds/linux/blob/master/include/misc/cxl.h
[2] https://github.com/ibm-capi/libcxl

CXL API

- CAPI accelerators are exposed via the Linux kernel's **cxl** [1] mechanism.
- IBM provides **libcxl** [2], a convenient user space wrapper library.
- Accelerator must signal completion by writing to one of the WED fields (and possibly raising an interrupt).
- Accelerator has *full access* to POWER8 user memory space (CAPI C/B/R).

```
// libcxl usage example
#include <libcxl.h>

// Prepare a work element descriptor.
struct {
  float *values;
  size_t num_values;
  uint64_t done;
} wed = { /* ... */ };
wed.done = 0;

// Offload work to accelerator.
struct cxl_afu_h *afu =
  cxl_afu_open_dev("/dev/cxl/afu0.0d");
cxl_afu_attach(afu, (uint64_t)&wed);
while (!wed.done); // better use
  interrupt
cxl_afu_free(afu);
```

[1] https://github.com/torvalds/linux/blob/master/include/misc/cxl.h
[2] https://github.com/ibm-capi/libcxl

CXL API

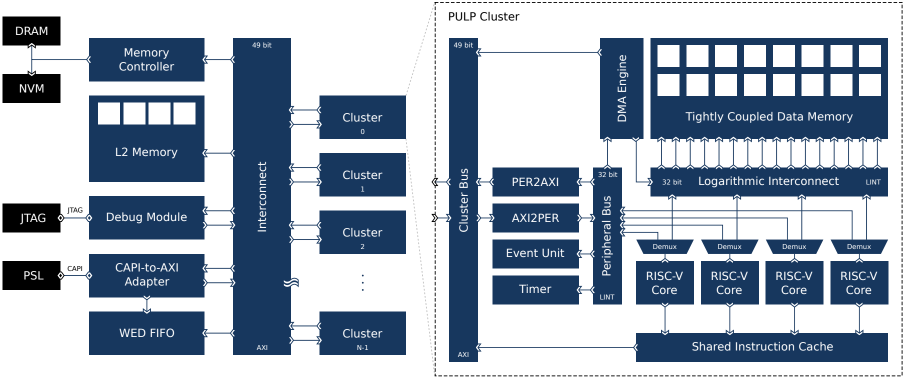
- CAPI accelerators are exposed via the Linux kernel's **cxl** [1] mechanism.
- IBM provides **libcxl** [2], a convenient user space wrapper library.
- Accelerator must signal completion by writing to one of the WED fields (and possibly raising an interrupt).
- Accelerator has *full access* to POWER8 user memory space (CAPI C/B/R).
- POWER8 has *limited access* to a few registers in the accelerator (CAPI MM).

```
// libcxl usage example
#include <libcxl.h>

// Prepare a work element descriptor.
struct {
  float *values;
  size_t num_values;
  uint64_t done;
} wed = { /* ... */ };
wed.done = 0;

// Offload work to accelerator.
struct cxl_afu_h *afu =
  cxl_afu_open_dev("/dev/cxl/afu0.0d");
cxl_afu_attach(afu, (uint64_t)&wed);
while (!wed.done); // better use
  interrupt
cxl_afu_free(afu);
```

[1] https://github.com/torvalds/linux/blob/master/include/misc/cxl.h
[2] https://github.com/ibm-capi/libcxl

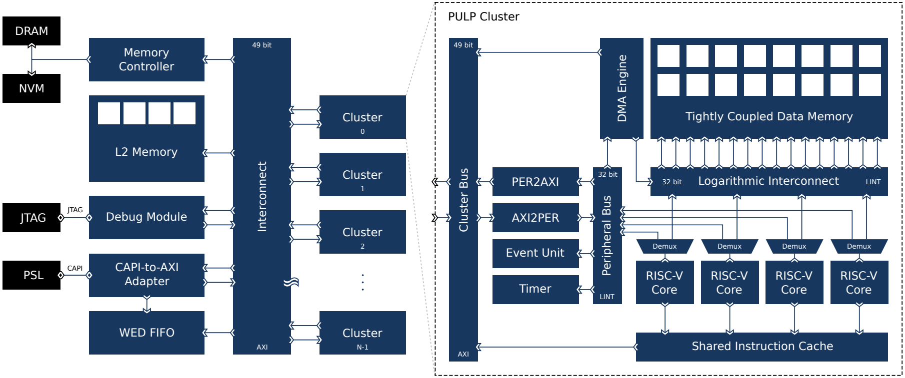
Integration of PULP and CAPI

- PULP system interconnect is AXI4-based



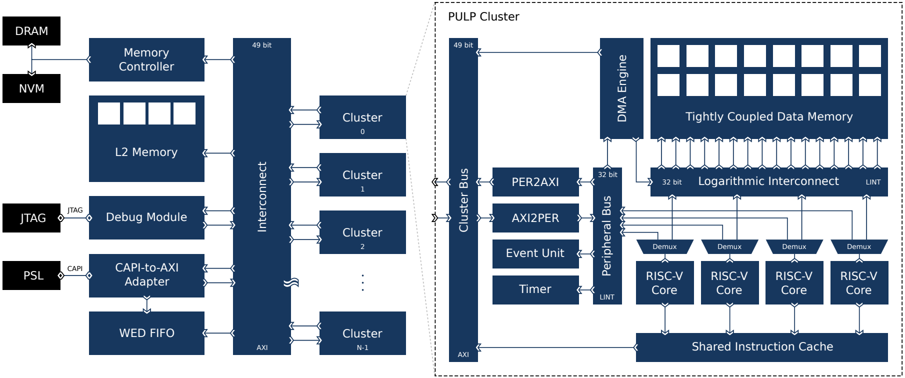
Integration of PULP and CAPI

- PULP system interconnect is AXI4-based
- *AXI-to-CAPI adapter* gives PULP access into POWER8 memory space

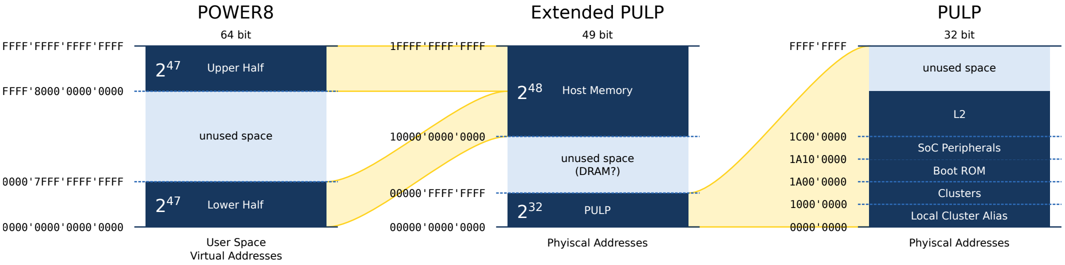


Integration of PULP and CAPI

- PULP system interconnect is AXI4-based
- *AXI-to-CAPI adapter* gives PULP access into POWER8 memory space
- Jobs from POWER8 are added to a *WED FIFO* where PULP can fetch them (woken up by interrupt)



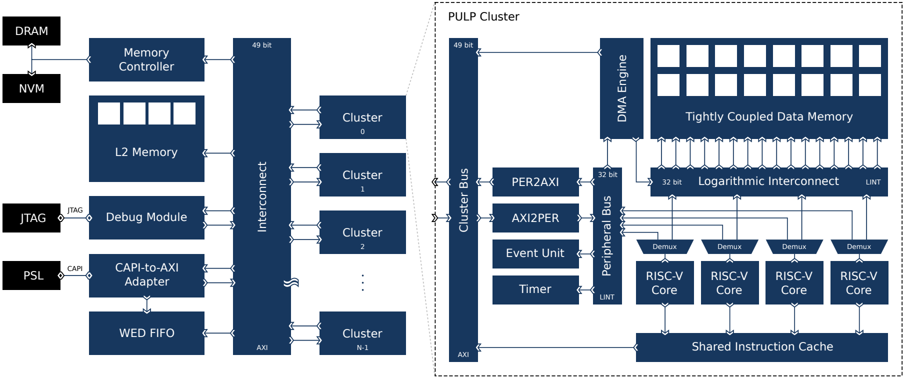
Memory Map



- POWER8 uses only lower 48 bits of address; upper bits sign-extension
- PULP itself is 32 bit (processors, clusters, peripherals)
- Selectively extend system-level interconnects to 49 bit
- MSB decides whether to access POWER8 or PULP memory
- Gained space in PULP memory can be used for on-board DRAM/NVM

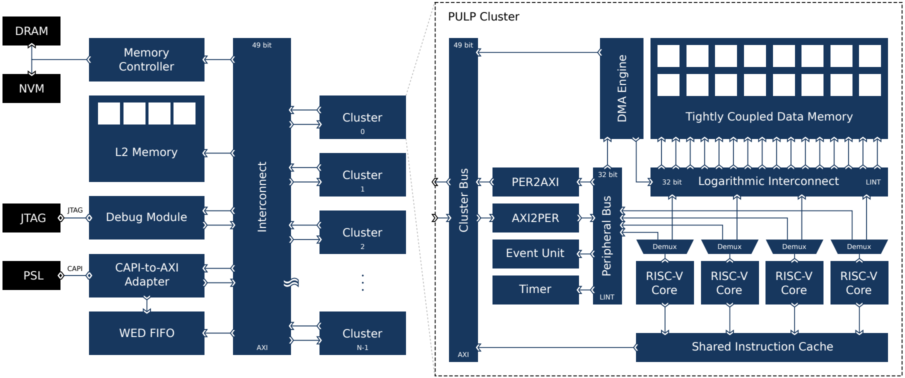
PULP 32 bit vs 64 bit

- PULP RISC-V cores and cluster peripherals are 32 bit



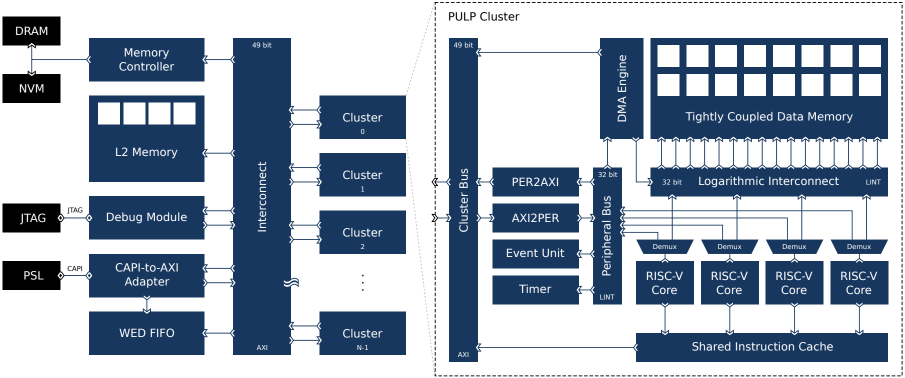
PULP 32 bit vs 64 bit

- PULP RISC-V cores and cluster peripherals are 32 bit
- DMA engine extended to support 64 bit



PULP 32 bit vs 64 bit

- PULP RISC-V cores and cluster peripherals are 32 bit
- DMA engine extended to support 64 bit
- **Caveat:** PULP cores cannot directly access POWER8 memory or DRAM; use DMA to copy data into cluster TCDM before crunching numbers



Offloading – How PULP Boots

- All cores start execution at the same internal address.
- Cores cannot directly execute code from host memory $\rightarrow$ kernel needs to be in PULP memory.
- Don't want to embed kernels into FPGA bitstream; would need to regenerate bitstream for every kernel change
- Embed a bootloader program into a ROM in the bitstream
- Send the PULP binary to execute with every WED
- Bootloader copies binary from POWER8 memory into PULP memory

Offloading – Binary Preparation on POWER8

- We want to be able to offload an ELF binary to PULP.

Offloading – Binary Preparation on POWER8

- We want to be able to offload an ELF binary to PULP.

1. Load the binary into memory.

Offloading – Binary Preparation on POWER8

- We want to be able to offload an ELF binary to PULP.

1. Load the binary into memory.
2. Parse the ELF header (`ehdr`) and program headers (`phdr`); these contain all sections that need to be loaded

Offloading – Binary Preparation on POWER8

- We want to be able to offload an ELF binary to PULP.

1. Load the binary into memory.
2. Parse the ELF header (`ehdr`) and program headers (`phdr`); these contain all sections that need to be loaded
3. Copy section offsets and sizes into a *section table*, and create a new WED:

```
struct wed {
  struct sec *sec_ptr;
  size_t sec_num;
  void *wed; // WED to be passed to loaded binary
};

struct sec {
  void *src;    // host memory
  uint32_t dst; // PULP memory
  uint32_t src_sz;
  uint32_t dst_sz;
};
```

Offloading – Binary Preparation on POWER8

- We want to be able to offload an ELF binary to PULP.

1. Load the binary into memory.
2. Parse the ELF header (`ehdr`) and program headers (`phdr`); these contain all sections that need to be loaded
3. Copy section offsets and sizes into a *section table*, and create a new WED:

```
struct wed {
  struct sec *sec_ptr;
  size_t sec_num;
  void *wed; // WED to be passed to loaded binary
};

struct sec {
  void *src;    // host memory
  uint32_t dst; // PULP memory
  uint32_t src_sz;
  uint32_t dst_sz;
};
```

4. Send to PULP as WED; bootloader then copies sections

Offloading – Bootloader

1. Only core 0 on cluster 0 (*fabric controller*) is active, other cores wait

Offloading – Bootloader

1. Only core 0 on cluster 0 (*fabric controller*) is active, other cores wait
2. Wait for a WED pointer (interrupt from job FIFO)

Offloading – Bootloader

1. Only core 0 on cluster 0 (*fabric controller*) is active, other cores wait
2. Wait for a WED pointer (interrupt from job FIFO)
3. Copy WED from POWER8 memory into the scratchpad with DMA

Offloading – Bootloader

1. Only core 0 on cluster 0 (*fabric controller*) is active, other cores wait
2. Wait for a WED pointer (interrupt from job FIFO)
3. Copy WED from POWER8 memory into the scratchpad with DMA
4. Copy the section table from POWER8 memory into the scratchpad with DMA

Offloading – Bootloader

1. Only core 0 on cluster 0 (*fabric controller*) is active, other cores wait
2. Wait for a WED pointer (interrupt from job FIFO)
3. Copy WED from POWER8 memory into the scratchpad with DMA
4. Copy the section table from POWER8 memory into the scratchpad with DMA
5. Copy every section in the table from POWER8 to PULP
   - Copy section in chunks into buffer in scratchpad with DMA
   - Write chunk to appropriate destination address in PULP memory space with DMA

Offloading – Bootloader

1. Only core 0 on cluster 0 (*fabric controller*) is active, other cores wait
2. Wait for a WED pointer (interrupt from job FIFO)
3. Copy WED from POWER8 memory into the scratchpad with DMA
4. Copy the section table from POWER8 memory into the scratchpad with DMA
5. Copy every section in the table from POWER8 to PULP
   - Copy section in chunks into buffer in scratchpad with DMA
   - Write chunk to appropriate destination address in PULP memory space with DMA
6. All cores jump to the start of the loaded binary

Offloading – liboprecomp API

We bundled the binary loading, parsing, and offloading code as a C library:

```
// liboprecomp

/* Binary loading and parsing */
opc_kernel_new          // create new kernel
opc_kernel_load_file    // parse binary from file
opc_kernel_load_buffer  // parse binary from memory
opc_kernel_free         // destroy kernel

/* Offloading onto PULP */
opc_dev_new             // create new device (= accelerator)
opc_dev_open_any        // open any device on the system
opc_dev_open_path       // open device by '/dev/cxl/...' path
opc_dev_launch          // offload kernel onto device
opc_dev_wait            // wait for completion of one kernel
opc_dev_wait_all        // wait for completion of all kernels
opc_dev_free            // destroy device
```

Wraps around *libcxl*, so this should be the only thing you need to interface with PULP.

Offloading – liboprecomp Usage Example

```
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary
3. Allocate new device

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary
3. Allocate new device
4. Open device

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary
3. Allocate new device
4. Open device
5. Offload kernel

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary
3. Allocate new device
4. Open device
5. Offload kernel
6. Wait for completion

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary
3. Allocate new device
4. Open device
5. Offload kernel
6. Wait for completion
7. Destroy device

Offloading – liboprecomp Usage Example

```c
// Error handling omitted for brevity.
// (Shame on me!)
#include <liboprecomp.h>

// Load the kernel.
const char *elf_path = "hello_world";
opc_kernel_t knl = opc_kernel_new();
opc_kernel_load_file(knl, elf_path);

// Open any accelerator on the system.
opc_dev_t dev = opc_dev_new();
opc_dev_open_any(dev);

// Offload a job and wait for completion.
uint64_t wed = 0xdeadbeeffacefeed;
opc_dev_launch(dev, knl, &wed, NULL);
opc_dev_wait_all(dev);

// Clean up.
opc_dev_free(dev);
opc_kernel_free(knl);
```

1. Allocate new kernel
2. Load kernel binary
3. Allocate new device
4. Open device
5. Offload kernel
6. Wait for completion
7. Destroy device
8. Destroy kernel

Example 1

Hello World

Let’s get started

Grab yourself a terminal, and:

```
cd; ./install.sh
```

Structure of the Examples

```
# tree
.
+-- host
|   +-- host.c
|   +-- Makefile
+-- Makefile
+-- pulp
    +-- Makefile
    +-- pulp.c

2 directories, 5 files
```

- `host`: contains the POWER8 code
- `pulp`: contains the PULP code
- `Makefile`:
  - calls `host/Makefile`
  - calls `pulp/Makefile`
  - emulates execution

Code – POWER8 Side

```
#include <liboprecomp.h>

int main(int argc, char **argv) {

  // Load the kernel.
  opc_kernel_t knl = opc_kernel_new();
  opc_kernel_load_file(knl, argv[1]);

  // Open any accelerator on the system.
  opc_dev_t dev = opc_dev_new();
  opc_dev_open_any(dev);

  // Offload a job and wait for completion.
  opc_dev_launch(dev, knl,
    (void*)0xdeadbeeffacefeed, NULL);
  opc_dev_wait_all(dev);

  // Clean up.
  opc_dev_free(dev);
  opc_kernel_free(knl);

  return 0;
}
```

1. Load PULP binary (argv[1])

Code – POWER8 Side

```c
#include <liboprecomp.h>

int main(int argc, char **argv) {

  // Load the kernel.
  opc_kernel_t knl = opc_kernel_new();
  opc_kernel_load_file(knl, argv[1]);

  // Open any accelerator on the system.
  opc_dev_t dev = opc_dev_new();
  opc_dev_open_any(dev);

  // Offload a job and wait for completion.
  opc_dev_launch(dev, knl,
    (void*)0xdeadbeeffacefeed, NULL);
  opc_dev_wait_all(dev);

  // Clean up.
  opc_dev_free(dev);
  opc_kernel_free(knl);

  return 0;
}
```

1. Load PULP binary (argv[1])
2. Connect to PULP on FPGA board

Code – POWER8 Side

```
#include <liboprecomp.h>

int main(int argc, char **argv) {

  // Load the kernel.
  opc_kernel_t knl = opc_kernel_new();
  opc_kernel_load_file(knl, argv[1]);

  // Open any accelerator on the system.
  opc_dev_t dev = opc_dev_new();
  opc_dev_open_any(dev);

  // Offload a job and wait for completion.
  opc_dev_launch(dev, knl,
    (void*)0xdeadbeeffacefeed, NULL);
  opc_dev_wait_all(dev);

  // Clean up.
  opc_dev_free(dev);
  opc_kernel_free(knl);

  return 0;
}
```

1. Load PULP binary (argv[1])
2. Connect to PULP on FPGA board
3. Run the computation

Code – POWER8 Side

```
#include <liboprecomp.h>

int main(int argc, char **argv) {

  // Load the kernel.
  opc_kernel_t knl = opc_kernel_new();
  opc_kernel_load_file(knl, argv[1]);

  // Open any accelerator on the system.
  opc_dev_t dev = opc_dev_new();
  opc_dev_open_any(dev);

  // Offload a job and wait for completion.
  opc_dev_launch(dev, knl,
    (void*)0xdeadbeeffacefeed, NULL);
  opc_dev_wait_all(dev);

  // Clean up.
  opc_dev_free(dev);
  opc_kernel_free(knl);

  return 0;
}
```

1. Load PULP binary (argv[1])
2. Connect to PULP on FPGA board
3. Run the computation
4. Offload kernel and wait (or do something else)

Code – POWER8 Side

```c
#include <liboprecomp.h>

int main(int argc, char **argv) {

  // Load the kernel.
  opc_kernel_t knl = opc_kernel_new();
  opc_kernel_load_file(knl, argv[1]);

  // Open any accelerator on the system.
  opc_dev_t dev = opc_dev_new();
  opc_dev_open_any(dev);

  // Offload a job and wait for completion.
  opc_dev_launch(dev, knl,
    (void*)0xdeadbeeffacefeed, NULL);
  opc_dev_wait_all(dev);

  // Clean up.
  opc_dev_free(dev);
  opc_kernel_free(knl);

  return 0;
}
```

1. Load PULP binary (argv[1])
2. Connect to PULP on FPGA board
3. Run the computation
4. Offload kernel and wait (or do something else)
5. Clean up

Code – PULP Side

```c
#include <stdint.h>
#include <stdio.h>

int main(uint64_t wed) {

  printf("Hello, World!\n");
  printf("You sent me me 0x%x\n", wed);

  return 0;
}
```

1. WED pointer passed to `main` as argument
2. A simple `printf` call
3. Print the pointer for reference (wed)

- Beware 64 bit and float caveats of the runtime for now!

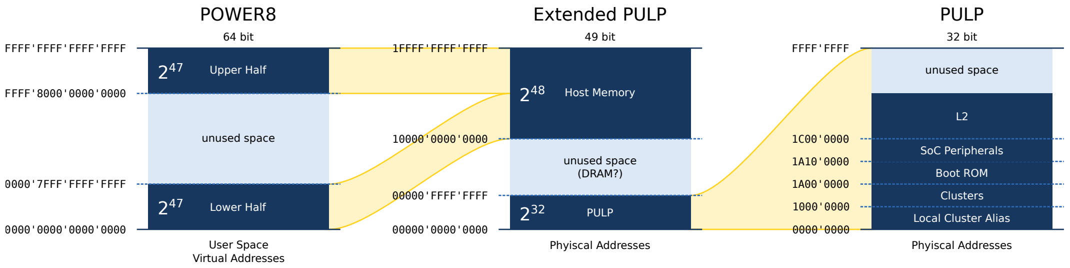
Your Turn!

1. **Goal:** Have PULP say hello
2. Boot into your virtual machine
3. **`cd`** `~/summerschool/ex1-hello`
4. Edit `host/host.c` and `pulp/pulp.c`
5. `make run`

Example 2

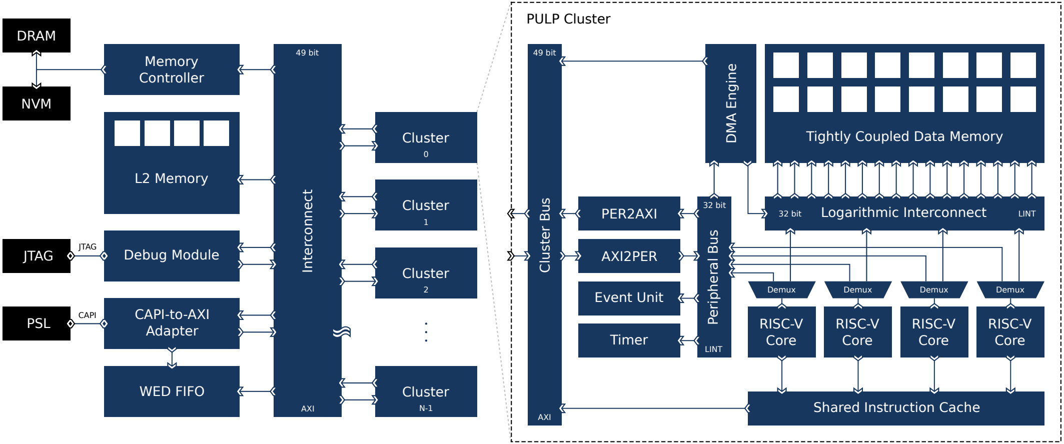
Data Movement

Memory Map



- PULP is 32 bit
- POWER8 is 64 bit
- POWER8 uses only lower 48 bits of address; upper bits sign-extension
- POWER8 memory mapped into upper part of 64 bit address space
- Cannot directly access its memory from PULP code
- **Solution:** Use the DMA!

Memory Hierarchy



- PULP cores operate directly on TCDM
- No direct access to host memory (32 bit limitation)
- DMA engine can do non-blocking copies
- L2 memory

Work Element Descriptors

- Only one 64 bit pointer can be passed to PULP
- memcpy requires at least:
  - source address
  - destination address
  - size of the block
- **Solution:** Work Element Descriptor
- Small struct prepared in POWER8 memory
- Contains all the information
- Pass WED pointer to PULP
- First step on PULP: Copy over WED from POWER8

```
// POWER8 side
struct wed {
  uint64_t num_words;
  float *input;
  float *output;
};

struct wed wed = { ... };
opc_dev_launch(dev, knl, &wed, NULL);
```

```
// PULP side
struct wed { ... };

struct wed wed;
int id = plp_dma_memcpy(
  host2local(wedptr), // remote
  (uint32_t)&wed,     // local
  sizeof(wed),        // size
  1                   // remote to local
);
plp_dma_wait(id);
```

Code – POWER8 / PULP

```
// Load binary
opc_kernel_new();
opc_kernel_load_file(...);

// Define WED
struct wed {
  uint64_t size;
  int64_t *input;
  volatile int64_t *output;
};
struct wed wed = { ... };

// Allocate input and output buffers
wed.input = calloc(...);
wed.output = calloc(...);

// Run PULP program.
opc_dev_new();
opc_dev_open_any(...);
opc_dev_launch(...);
opc_dev_wait_all(...);

// Check the results.
wed.input[i] == wed.output[i];
```

```
// PULP side
// Load Work Element Descriptor
struct wed wed;
plp_dma_memcpy(
  host2local(wedptr), // remote
  (uint32_t)&wed,     // local
  sizeof(wed),        // size
  1                   // remote to local
);

// Allocate a local buffer on PULP to
  hold the data.
void *buffer = malloc(wed.size);

// Copy data from host to buffer and
  back to host.
plp_dma_memcpy(..., 1);
plp_dma_wait(...);
plp_dma_memcpy(..., 0);
plp_dma_wait(...);
```

Your Turn!

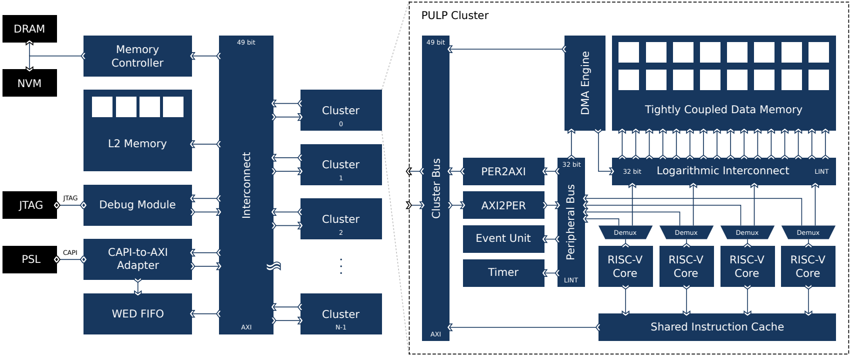
1. **Goal:** Offload `memcpy` to PULP
2. **`cd`** `~/summerschool/ex2-dma`
3. Edit `host/host.c` and `pulp/pulp.c`
4. `make run`

Example 3

Simple Computation

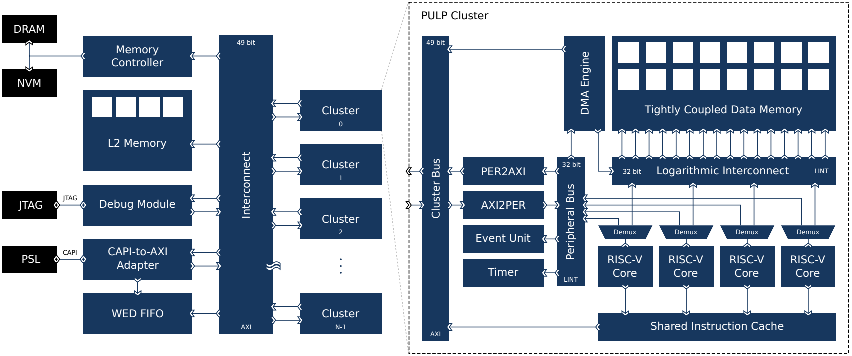
Let’s do some computation

- Last example moved data through PULP
- Opportunity to do some computation while we have the data
- Memory size limited; **how can we handle arbitrary amounts of data?**



Tiling

- Memory limited (64 kB L1, 256 kB L2)
- Data does not fit into fast small memories
- **Solution:** Divide input data into tiles
- Operate tile-by-tile
- In CPUs/GPUs the cache implicitly does this (but costs energy!)



Your Turn!

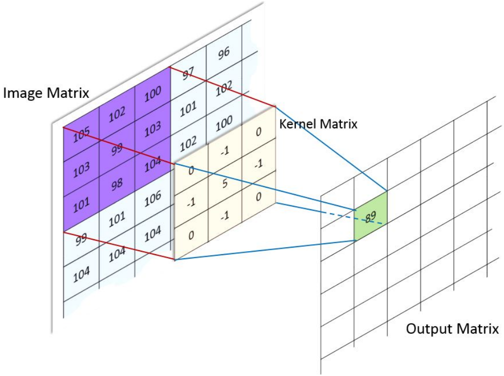
1. **Goal:** Implement a kernel that squares each value in an arbitrarily sized buffer
2. `cd ~/summerschool/ex3-square`
3. Edit `host/host.c` and `pulp/pulp.c`
4. `make run`

Example 4

Convolution

Convolution in 2D

- Very popular in Deep Neural Networks
- Usually: Apply a filter with local response to an image
- A kind of stencil operation
- $y = x * w$
- For example: Edge detection



$$w_{u,v} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

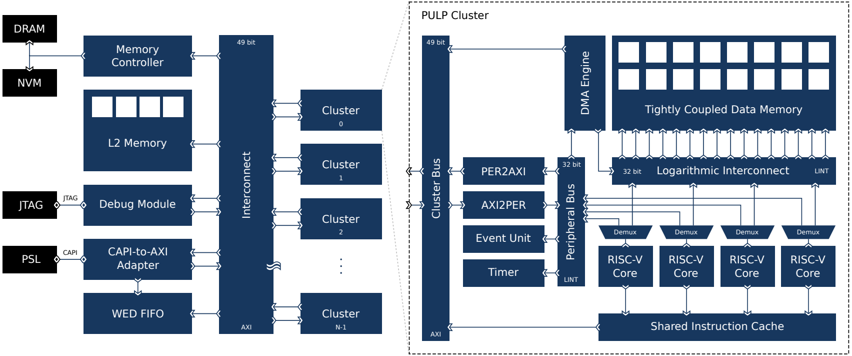
Code – Simple

- Input/output image has 3 dimensions:
  - width M
  - height N
  - channels K
- Filter kernel has 2 dimensions
  - width V
  - height U
- 1 output pixel influenced by 9 input pixels
- Strategy:
  - Iterate over each output pixel (K,N,M)
  - Multiply-add pixels in the neighborhood (U,V)
- **Careful about zero-padding!**

```
float x[K][N][M];
float w[U][V];
float y[K][N][M];

for (int k = 0; k < M; ++k)
for (int n = 0; n < N; ++n)
for (int m = 0; m < M; ++m) {
  float a = 0f;
  for (int u = 0; u < U; ++u)
  for (int v = 0; v < V; ++v) {
    int n_ = n+u-U/2;
    int m_ = m+v-V/2;
    if (n_ < 0 || m_ < 0 || n_ >= N || m_ >= M)
      continue;
    a += x[n_][m_][k] * w[u][v];
  }
  y[n][m][k] = a;
}
```

Tiling

- Memory limited (64 kB L1, 256 kB L2)
- Data does not fit into fast small memories
- **Solution:** Divide input image into 2D tiles
- Operate tile-by-tile
- In CPUs/GPUs the cache implicitly does this (but costs energy!)



Code – Tiled

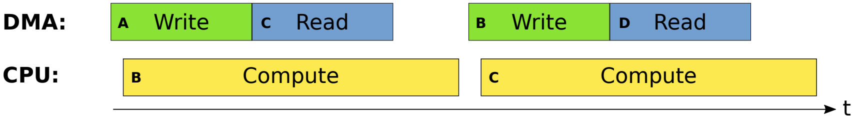
- Split image dimensions into tiles that fit into memory
- Tile size TS
- Process tile-by-tile
- Use the DMA's 2D transfer capability
- Assume image a multiple of the tile size

```
// ...
const int TS = 64; // tile size

for (int k = 0; k < M; ++k)
for (int n1 = 0; n1 < N/TS; ++n1)
for (int m1 = 0; m1 < M/TS; ++m1) {
  // Load tile here
  for (int n2 = 0; n2 < TS; ++n2)
  for (int m2 = 0; m2 < TS; ++m2) {
    float a = 0f;
    for (int u = 0; u < U; ++u)
    for (int v = 0; v < V; ++v) {
      int n_ = n1*TS + n2 + u-U/2;
      int m_ = m1*TS + m2 + v-V/2;
      // ...
    }
    y[n][m][k] = a;
  }
  // Store tile here
}
```

Double Buffering

- Overlay data movement and computation
- Hides latency of memory system
- Implicit in GPUs/CPUs, explicit in PULP
- Recipe:

1. Load input data (background)
2. Block and wait for DMA
3. Trigger last write (background)
4. Compute
5. Schedule write of output data



Double Buffering – Implementation

```c
static struct {
  void *src;
  uint64_t dst;
  size_t size;
} writeback = {
  .src = NULL,
  .dst = 0,
  .size = 0
};

void writeback_schedule(
  void *src,
  uint64_t dst,
  size_t size
) {
  writeback_trigger();
  writeback.src = src;
  writeback.dst = dst;
  writeback.size = size;
}
```

```c
void writeback_trigger() {
  if (writeback.size == 0) return;
  plp_dma_memcpy(
    host2local(writeback.dst),
    (uint32_t)writeback.src,
    writeback.size,
    PLP_DMA_LOC2EXT
  );
  writeback.size = 0;
}

// In your code:
for (int i = 0; i < N; ++i) {
  // load tile (1)
  plp_dma_barrier(); // (2)
  writeback_trigger(); // start write-back (3)
  // do computation (4)
  writeback_schedule(...); // schedule write-back (5)
}
writeback_trigger(); // final write-back
plp_dma_barrier();
```

Your Turn!

1. **Goal:** Implement a convolution filter
2. Use tiling to fit into memory
3. Use double buffering to hide latency
4. `cd ~/summerschool/ex4-conv`
5. Edit `host/host.c` and `pulp/pulp.c`
6. `make run`

Coming Soon

- Multicore & multicluster examples
- `float{8,16,16alt}` support in the virtual platform
- Transprecision unit in the actual hardware (FPGA bitstream)
- More examples as benchmarks get ported

- We're happy to help:

`fschuiki@iis.ee.ethz.ch`
`smach@iis.ee.ethz.ch`

- We'd love your input on API improvements!

Thanks! Questions?

OPRECOMP
Open Transprecision Computing