Hardware acceleration for PyTorch – a demonstrator

with Embecosm

The team

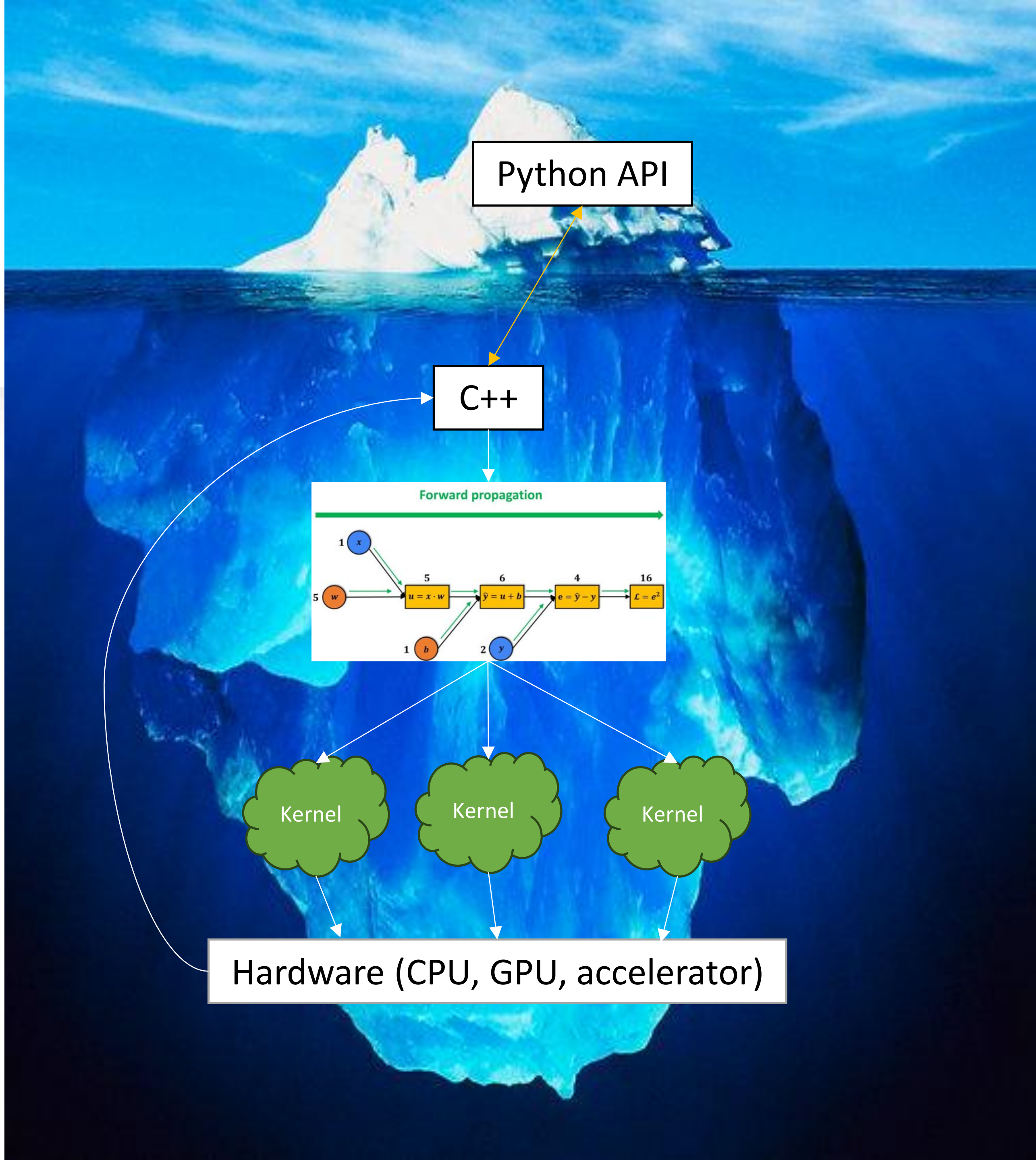
MEng students at the University of Southampton.

- James Betson (electronic engineering)
- Lorinc Boer (computer science)
- Reuben Leanage (electronic engineering)
- Tao Zeng (electronic engineering)
- Ze Yong (electronic engineering)
- AoYang Leng (electronic engineering)

PyTorch

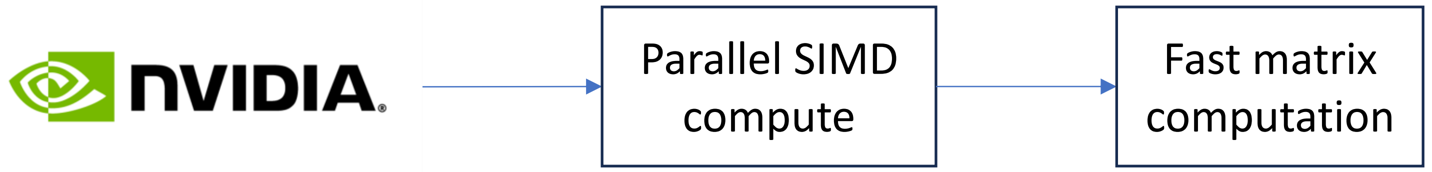
- Python API for building **neural networks**.
- Built on top of C++ using pybind.

- Used by Meta, Apple, TikTok, Microsoft…
- **Quick** and **easy**.



A fly in the ointment?

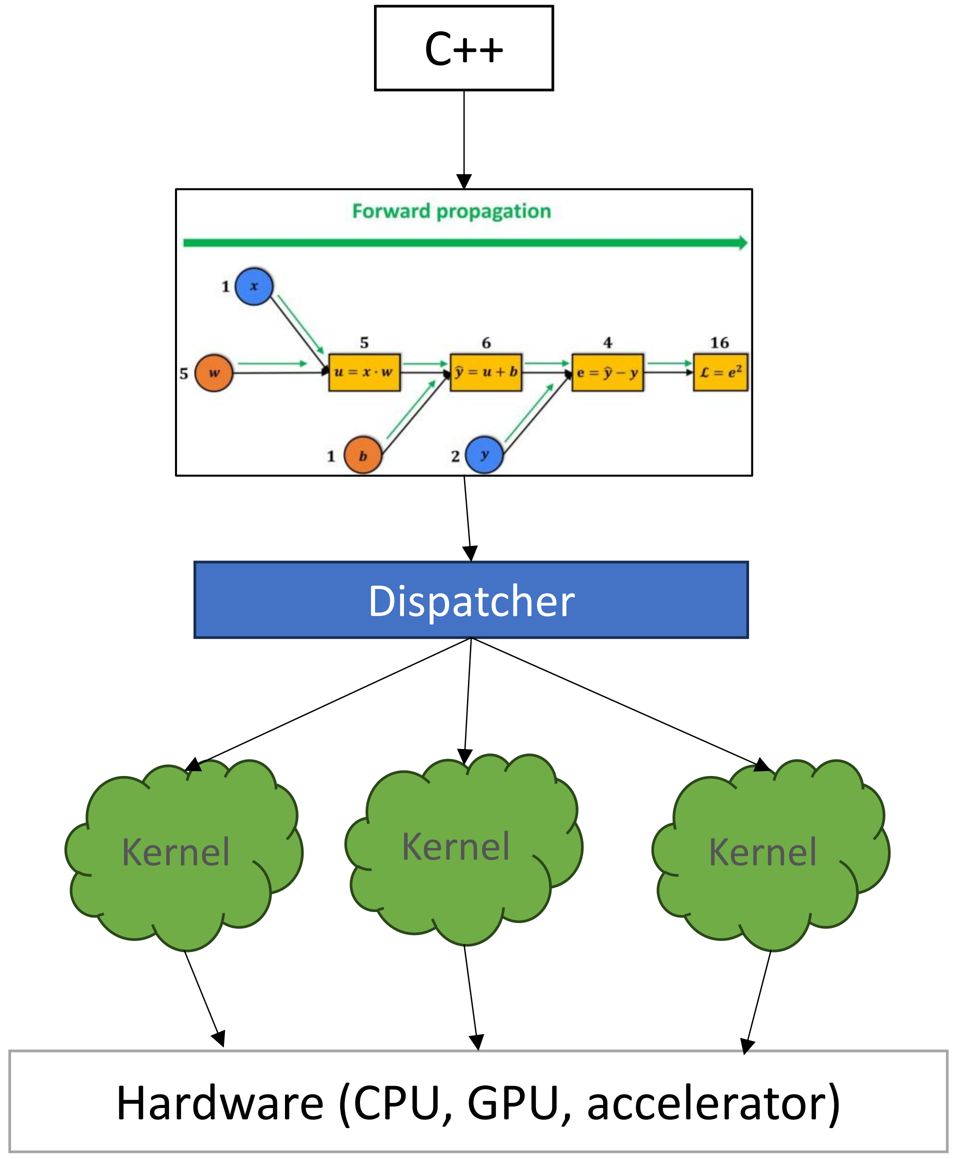
- PyTorch has great support for CPUs and GPUs.



- Support for custom hardware? What about a bespoke ASIC or FPGA design?

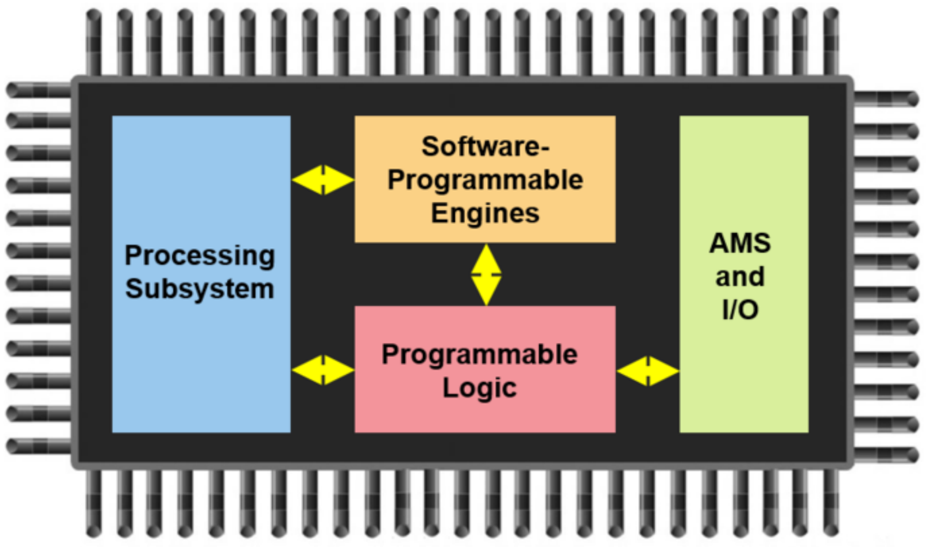
A fly in the ointment?

- Hardware dependent **kernels**



Custom accelerators

- Improved Performance.
- Specialized hardware.
- Unlimited flexibility.



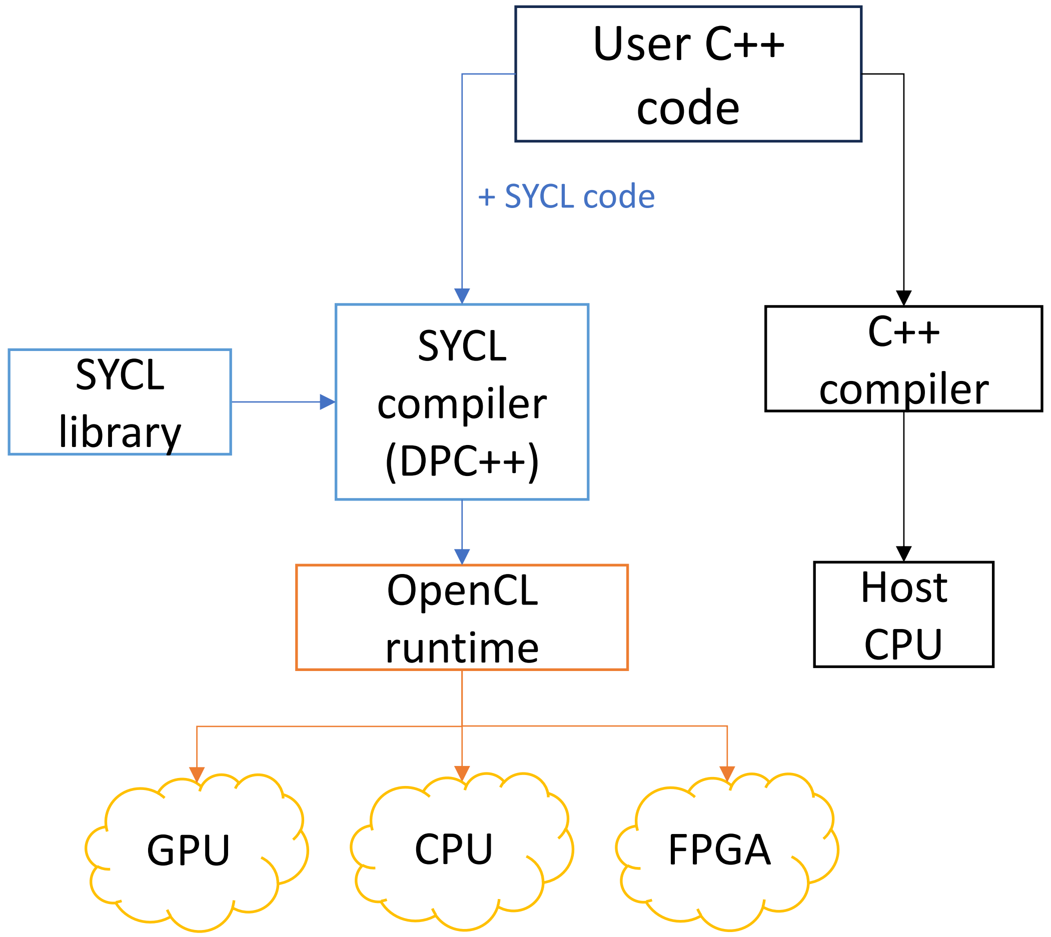
Heterogeneous Computing Platform

The project

"One of Pytorch's current weaknesses is how **difficult it is to add support for new accelerators** in Eager mode. As it stands Pytorch defines several thousand core operations, each of which must be implemented individually for any new hardware – **an extremely large undertaking**."

- Investigate how this can be done.
- Build a proof-of-concept demonstrator.

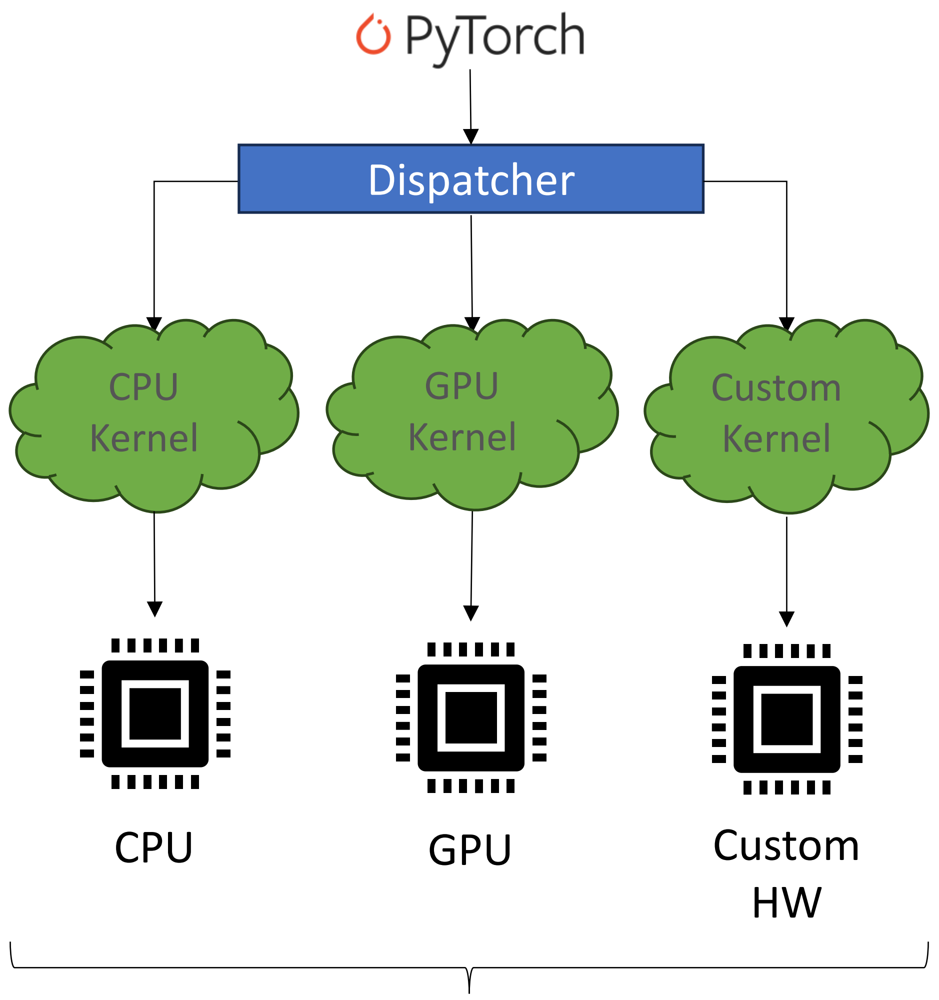
SYCL: How does it work?



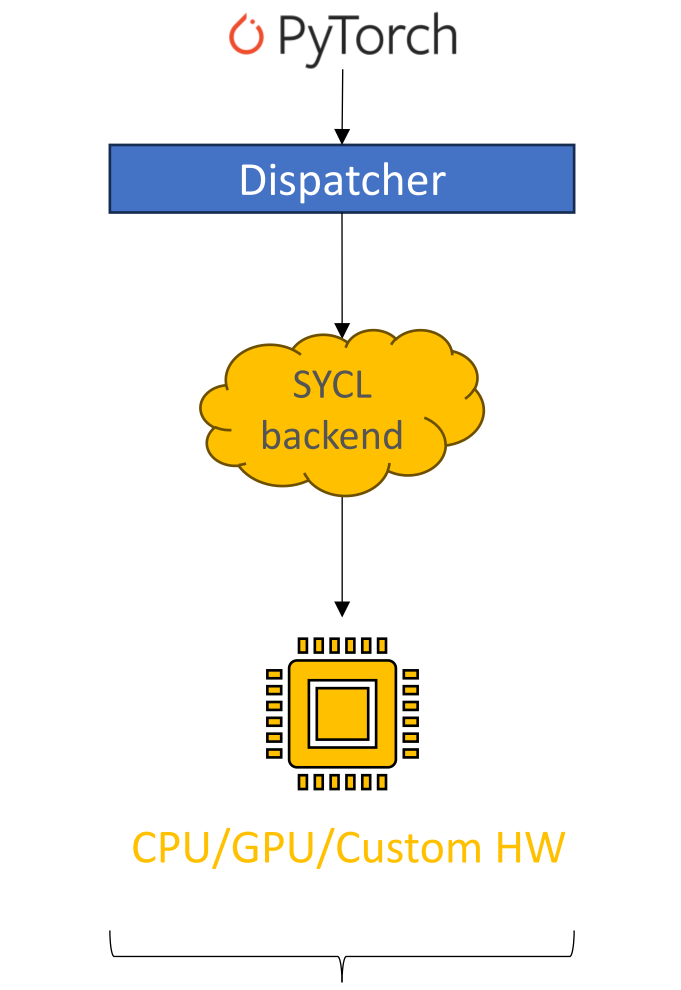
- C++ programming model
- Generic hardware-agnostic (abstraction)

- Heterogeneous compute framework
- Interfaces with hardware

Dispatch Comparison



Typical PyTorch offload



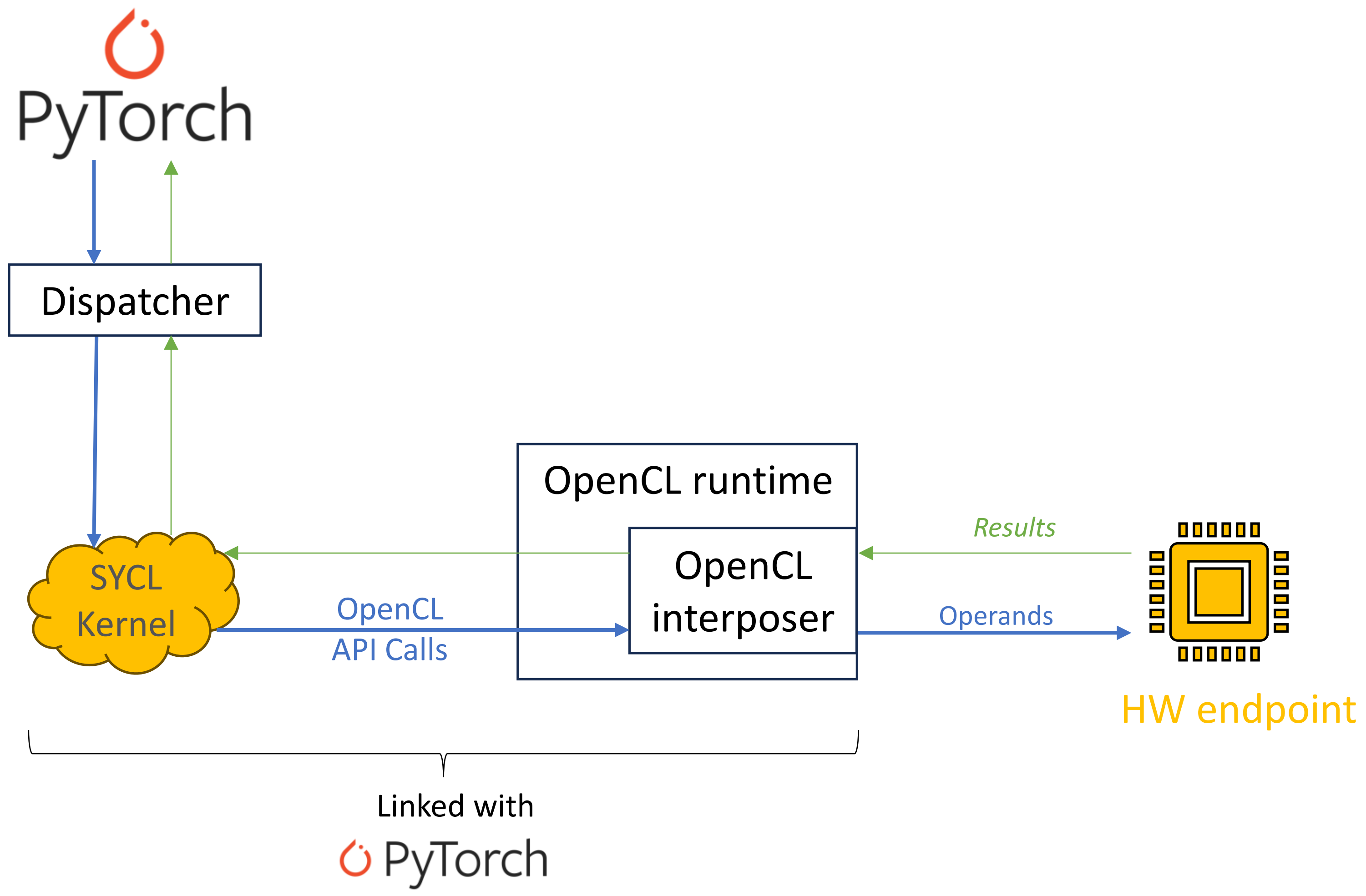
(Theoretical) PyTorch SYCL offload

C++ extensions

- PyTorch tool (pybind).
  - Only supports a standard g++ compilation process.

Solution: write standard C++ kernels
- Call SYCL functions as '`extern`'.
- Compile SYCL functions separately into a **shared object.**
- Link: '`export LD_PRELOAD=$(pwd)/build/libsycl_library.so`'

Our Implementation



- C++ programming model
- Generic hardware-agnostic (abstraction)

- Heterogeneous compute framework
- Interfaces with hardware

Interposer – why?

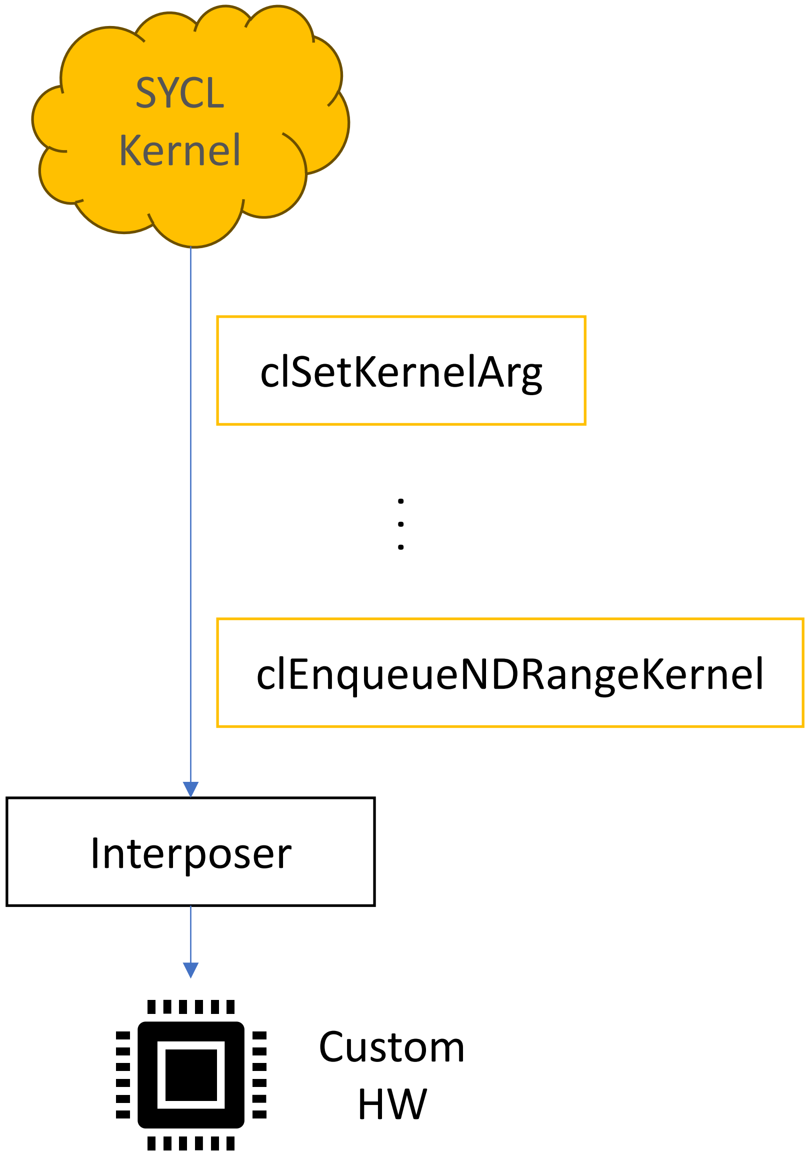
- oneAPI HAL: struggled to bring up. Worried about overhead.
- OpenCL device: would take too long. Lots of overhead.

Solution: manually intercept OpenCL calls to '*sycl::default_selector_v*'.

- Extract operands and operation type.

Interposer – how?

1. clSetKernelArg – captures SYCL data buffer
2. …
3. …
4. …
5. …
6. …
7. …
8. clEnqueueNDRangeKernel – trigger computation



Why is this better?

- No more re-writing PyTorch source code!

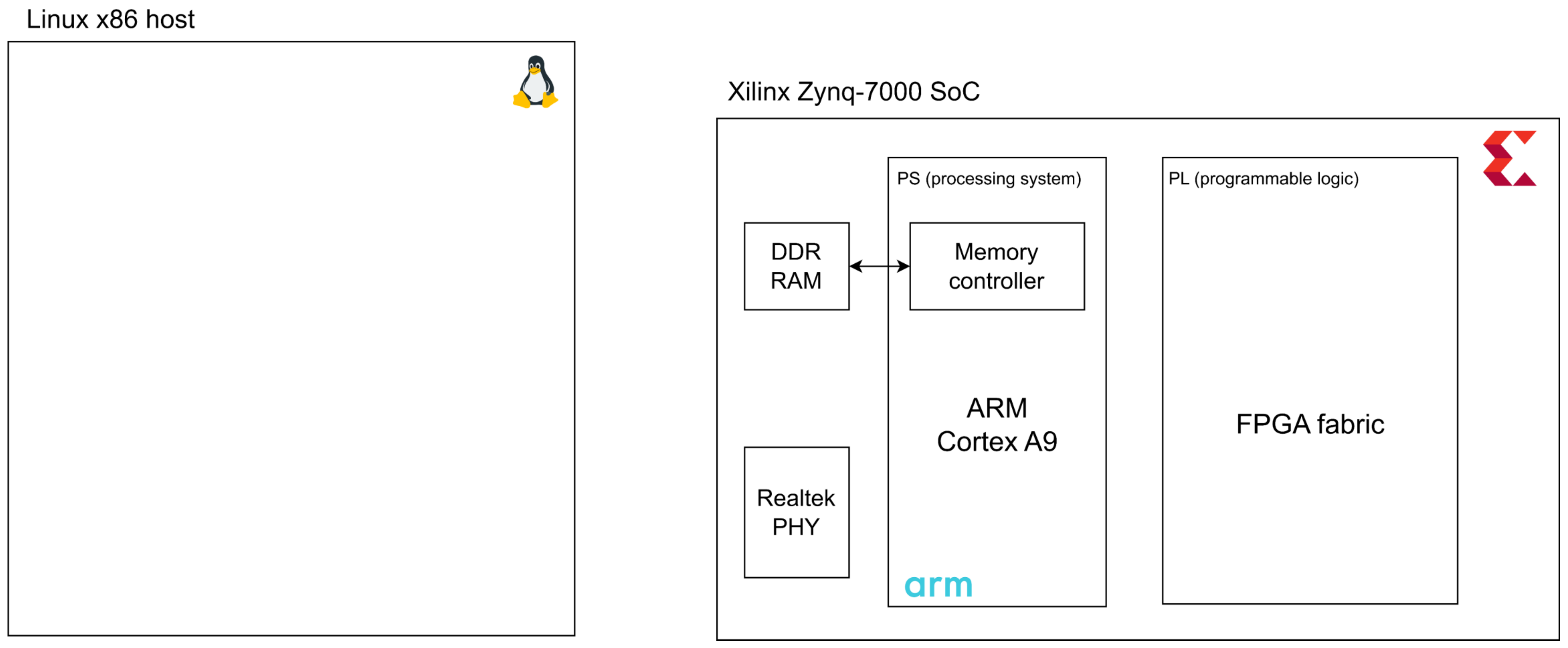
- User engineering effort has changed:

Before	After
Re-write potentially 1000s of kernels – **hours of work**	Use PyTorch as is – **easy**
Repeat this process and re-compile every time hardware changes – **tedious**	Interface with OpenCL API calls. Easy to intercept data for your hardware to run

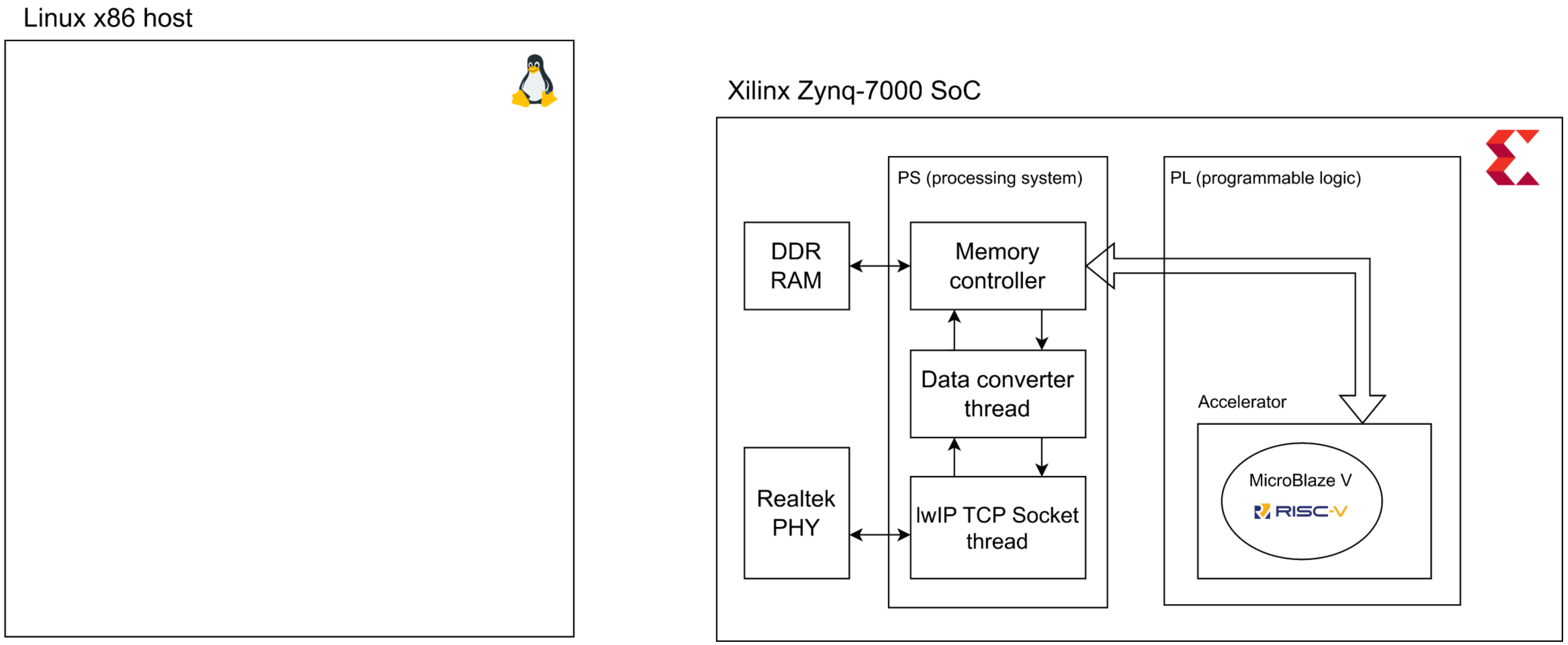
Why is this better

- Shifting user effort **away** from PyTorch.
- AI tools must keep up!

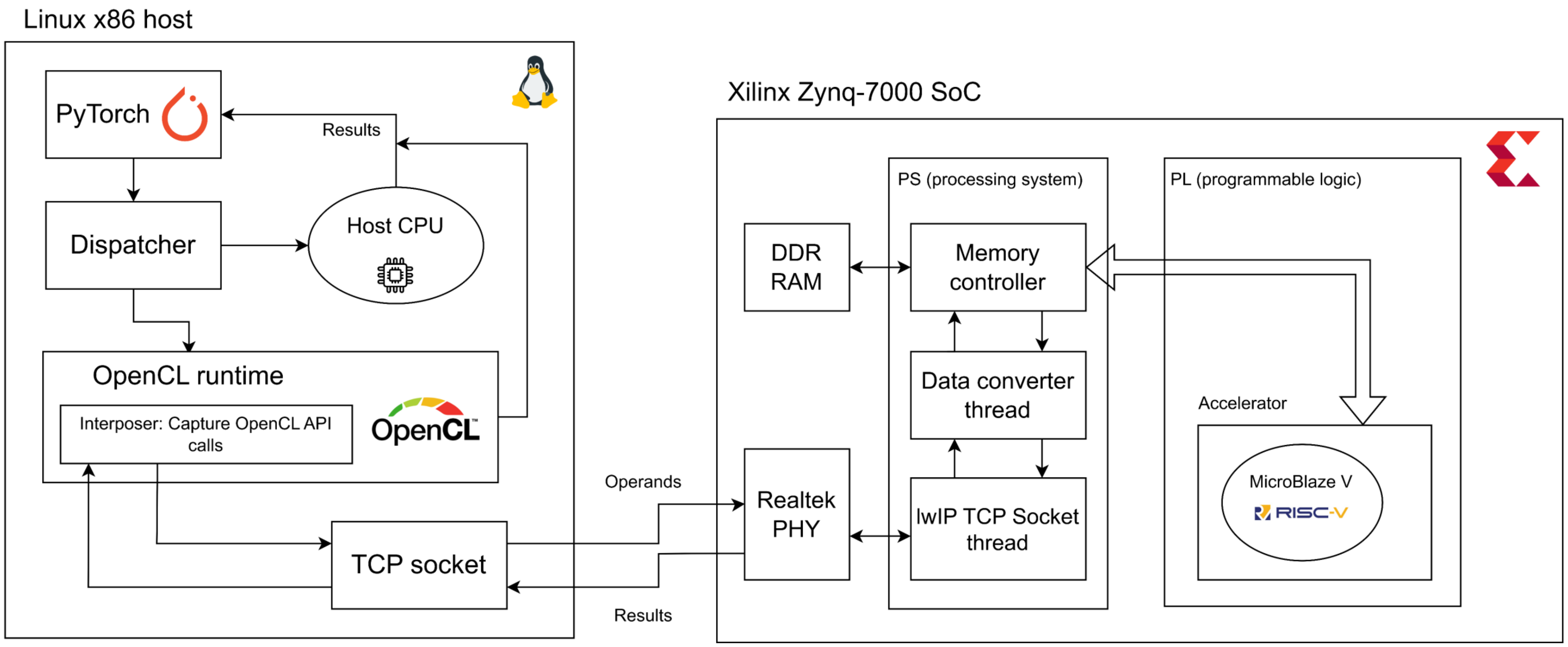
Our demonstrator



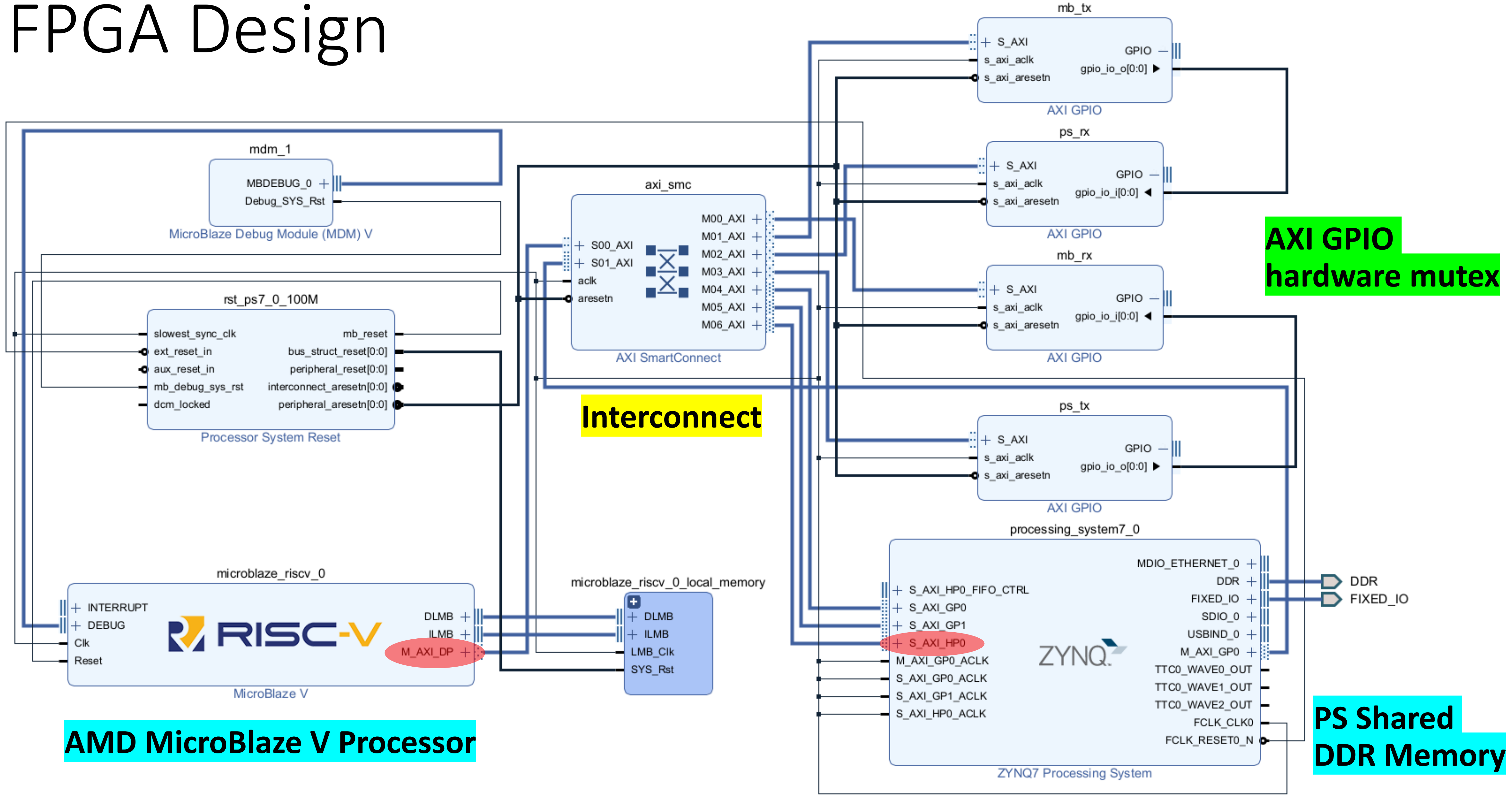
Our demonstrator



Our demonstrator

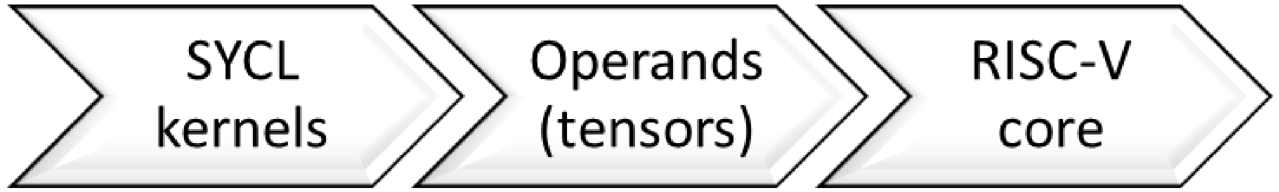


FPGA Design


mb_tx
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_o[0:0]
AXI GPIO
ps_rx
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_i[0:0]
AXI GPIO
mb_rx
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_i[0:0]
AXI GPIO
ps_tx
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
gpio_io_o[0:0]
AXI GPIO
AXI GPIO hardware mutex
mdm_1
MBDEBUG_0
Debug_SYS_Rst
MicroBlaze Debug Module (MDM) V
axi_smc
S00_AXI
S01_AXI
aclk
aresetn
M00_AXI
M01_AXI
M02_AXI
M03_AXI
M04_AXI
M05_AXI
M06_AXI
AXI SmartConnect
Interconnect
rst_ps7_0_100M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
microblaze_riscv_0
INTERRUPT
DEBUG
Clk
Reset
RISC-V
DLMB
ILMB
M_AXI_DP
MicroBlaze V
AMD MicroBlaze V Processor
microblaze_riscv_0_local_memory
DLMB
ILMB
LMB_Clk
SYS_Rst
processing_system7_0
S_AXI_HP0_FIFO_CTRL
S_AXI_GP0
S_AXI_GP1
S_AXI_HP0
M_AXI_GP0_ACLK
S_AXI_GP0_ACLK
S_AXI_GP1_ACLK
S_AXI_HP0_ACLK
ZYNQ
MDIO_ETHERNET_0
DDR
FIXED_IO
SDIO_0
USBIND_0
M_AXI_GP0
TTC0_WAVE0_OUT
TTC0_WAVE1_OUT
TTC0_WAVE2_OUT
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System
DDR
FIXED_IO
PS Shared DDR Memory

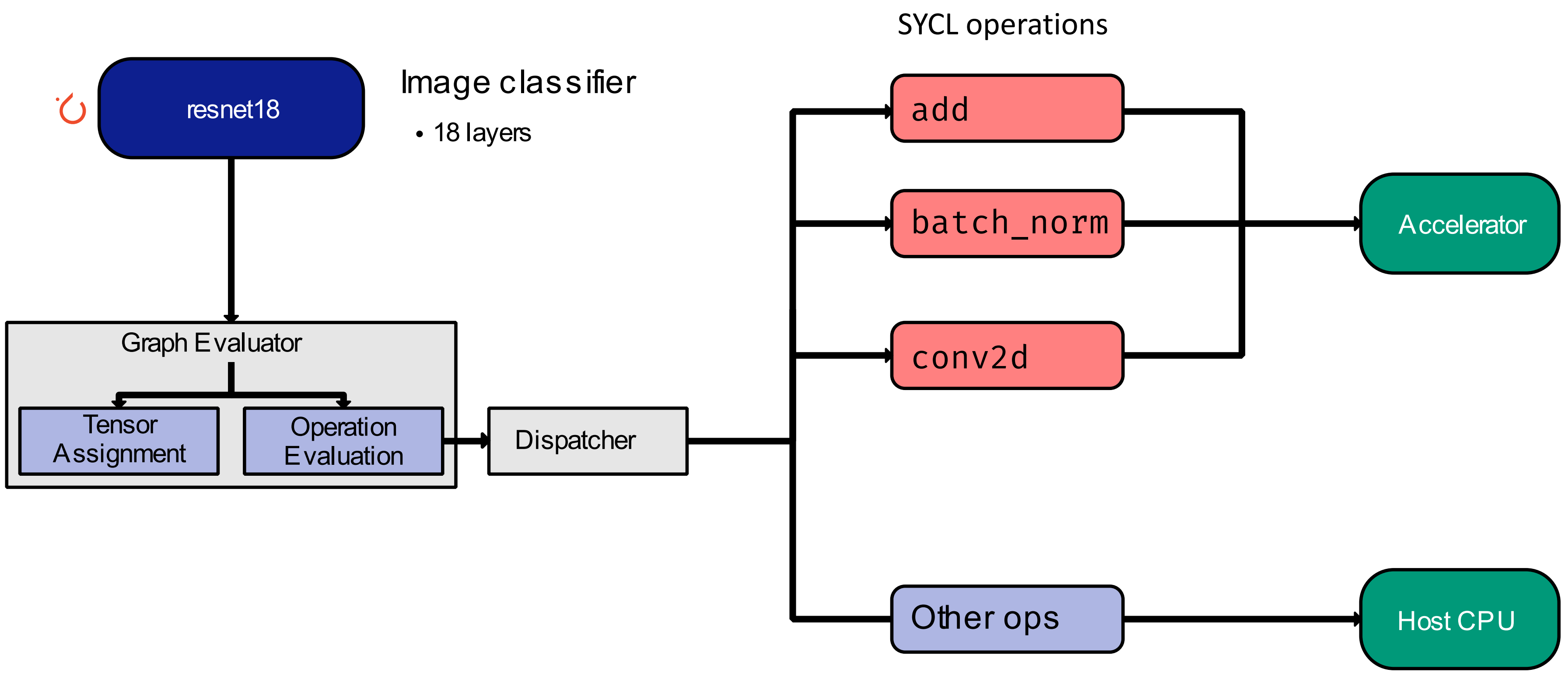
Proof-of-Concept

- RISC-V soft core = accelerator
- Data flow:

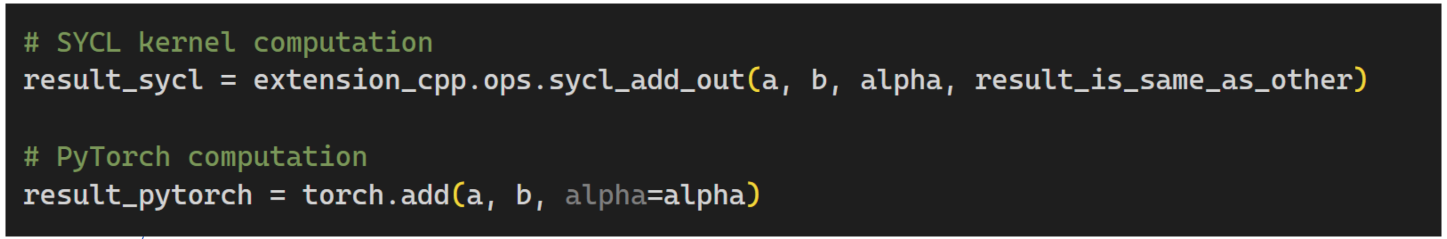


- Show that we can offload compute from PyTorch to an accelerator
  - *Without* the user re-writing the PyTorch backends!
  - No re-writing for *new* accelerators – **scalable, hardware-agnostic**

Proof-of-concept

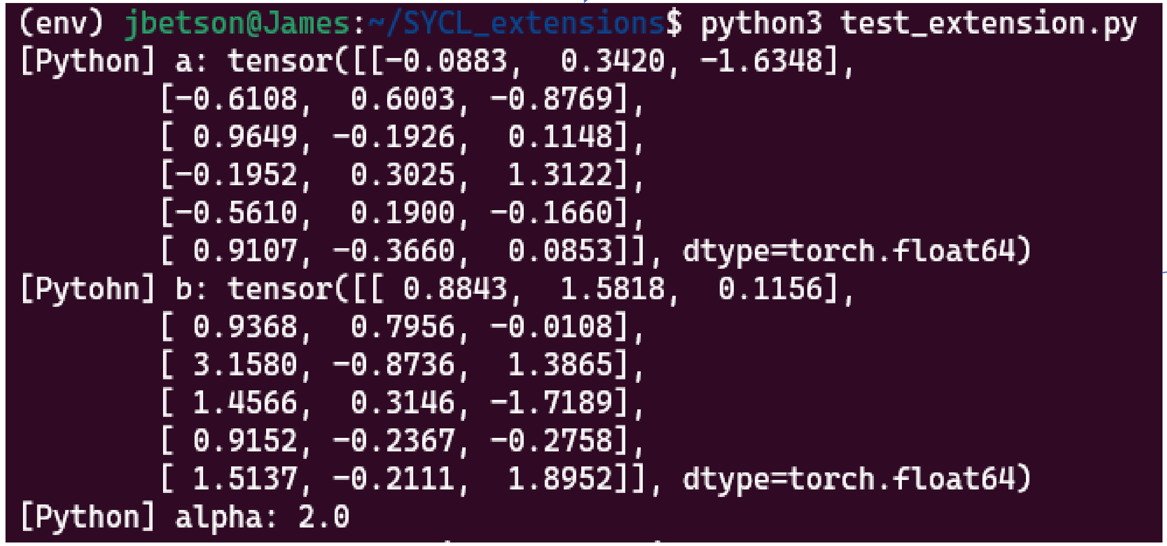


Results

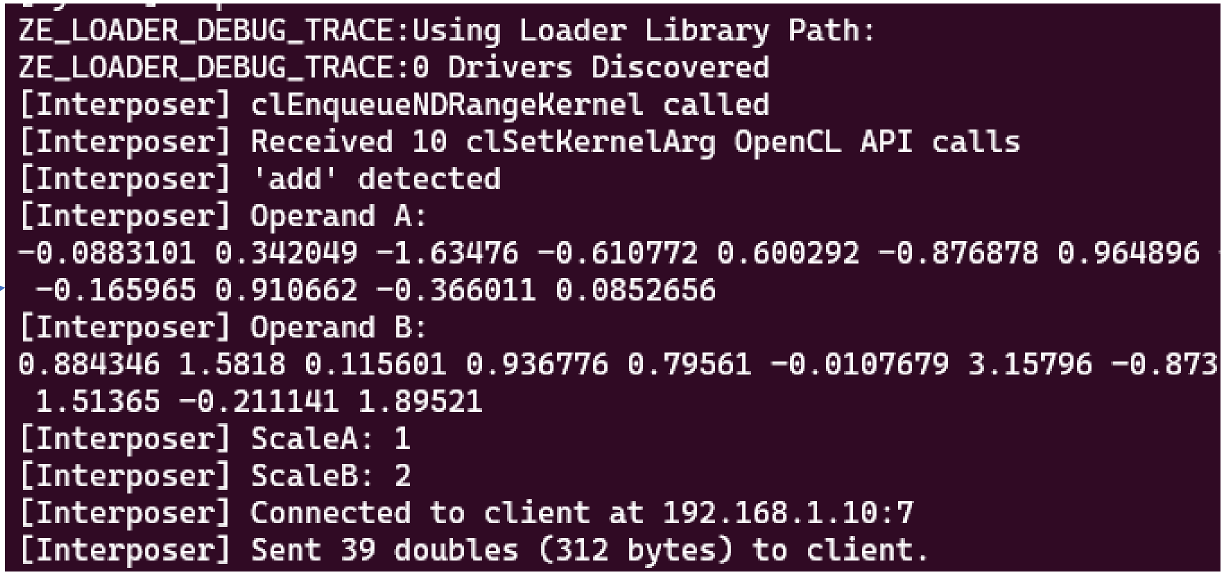


```
# SYCL kernel computation
result_sycl = extension_cpp.ops.sycl_add_out(a, b, alpha, result_is_same_as_other)

# PyTorch computation
result_pytorch = torch.add(a, b, alpha=alpha)
```

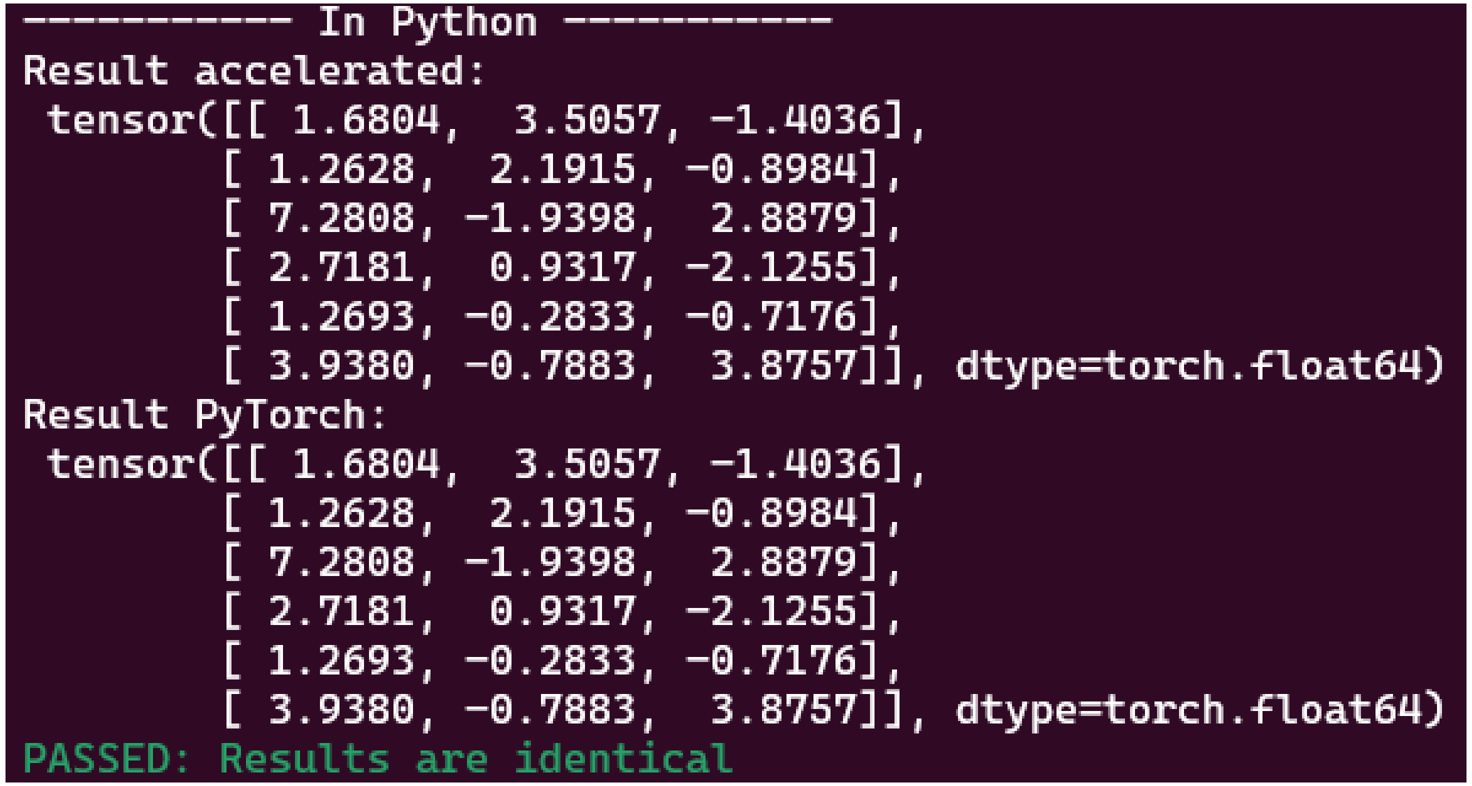


```
(env) jbetson@James:~/SYCL_extensions$ python3 test_extension.py
[Python] a: tensor([[-0.0883,  0.3420, -1.6348],
        [-0.6108,  0.6003, -0.8769],
        [ 0.9649, -0.1926,  0.1148],
        [-0.1952,  0.3025,  1.3122],
        [-0.5610,  0.1900, -0.1660],
        [ 0.9107, -0.3660,  0.0853]], dtype=torch.float64)
[Pytohn] b: tensor([[ 0.8843,  1.5818,  0.1156],
        [ 0.9368,  0.7956, -0.0108],
        [ 3.1580, -0.8736,  1.3865],
        [ 1.4566,  0.3146, -1.7189],
        [ 0.9152, -0.2367, -0.2758],
        [ 1.5137, -0.2111,  1.8952]], dtype=torch.float64)
[Python] alpha: 2.0
```



```
ZE_LOADER_DEBUG_TRACE:Using Loader Library Path:
ZE_LOADER_DEBUG_TRACE:0 Drivers Discovered
[Interposer] clEnqueueNDRangeKernel called
[Interposer] Received 10 clSetKernelArg OpenCL API calls
[Interposer] 'add' detected
[Interposer] Operand A:
-0.0883101 0.342049 -1.63476 -0.610772 0.600292 -0.876878 0.964896
 -0.165965 0.910662 -0.366011 0.0852656
[Interposer] Operand B:
0.884346 1.5818 0.115601 0.936776 0.79561 -0.0107679 3.15796 -0.873
 1.51365 -0.211141 1.89521
[Interposer] ScaleA: 1
[Interposer] ScaleB: 2
[Interposer] Connected to client at 192.168.1.10:7
[Interposer] Sent 39 doubles (312 bytes) to client.
```

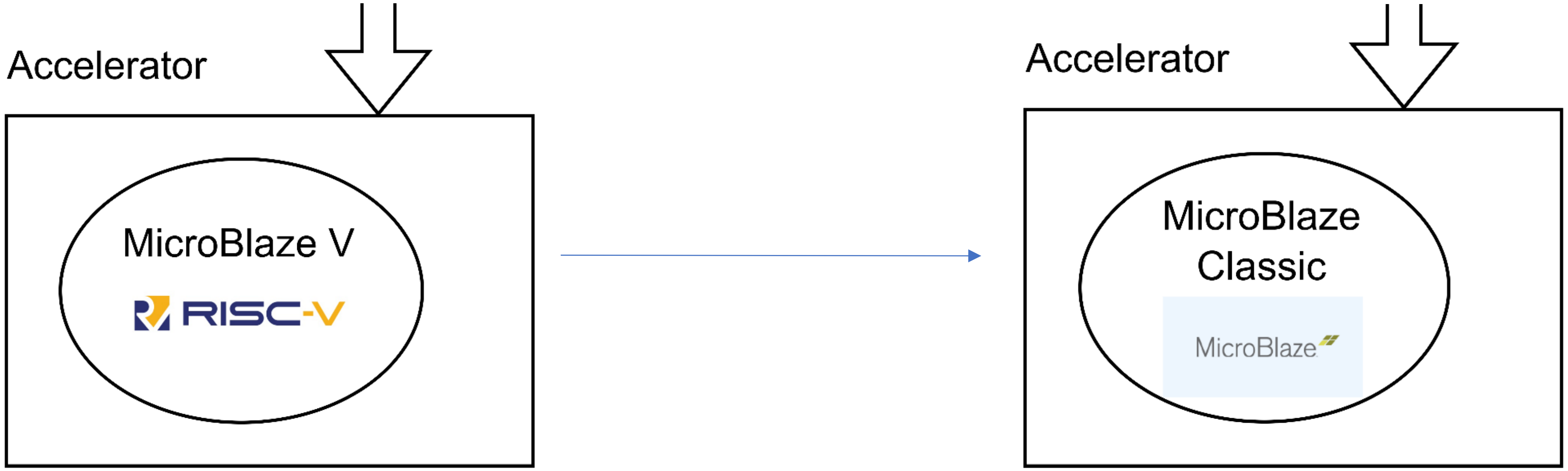
Results



```
----------- In Python -----------
Result accelerated:
 tensor([[ 1.6804,  3.5057, -1.4036],
        [ 1.2628,  2.1915, -0.8984],
        [ 7.2808, -1.9398,  2.8879],
        [ 2.7181,  0.9317, -2.1255],
        [ 1.2693, -0.2833, -0.7176],
        [ 3.9380, -0.7883,  3.8757]], dtype=torch.float64)
Result PyTorch:
 tensor([[ 1.6804,  3.5057, -1.4036],
        [ 1.2628,  2.1915, -0.8984],
        [ 7.2808, -1.9398,  2.8879],
        [ 2.7181,  0.9317, -2.1255],
        [ 1.2693, -0.2833, -0.7176],
        [ 3.9380, -0.7883,  3.8757]], dtype=torch.float64)
PASSED: Results are identical
```

- For input tensors up to 512 doubles.
- Supports ‘add’ and ‘batch_norm’.
- Insufficient time for ‘conv2d’ but SYCL support was built.

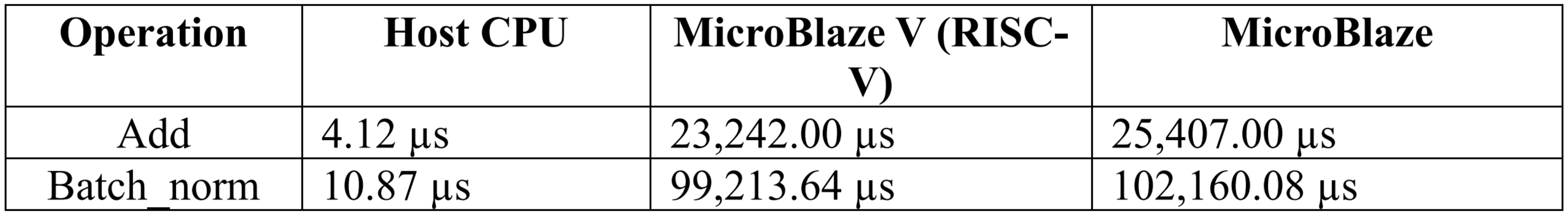
Case study: portability



Case study: performance

- Experimental results: ~5600 times slower for ‘add’ | ~9100 times slower for ‘batch_norm’.

Table 7 – Final design performance results



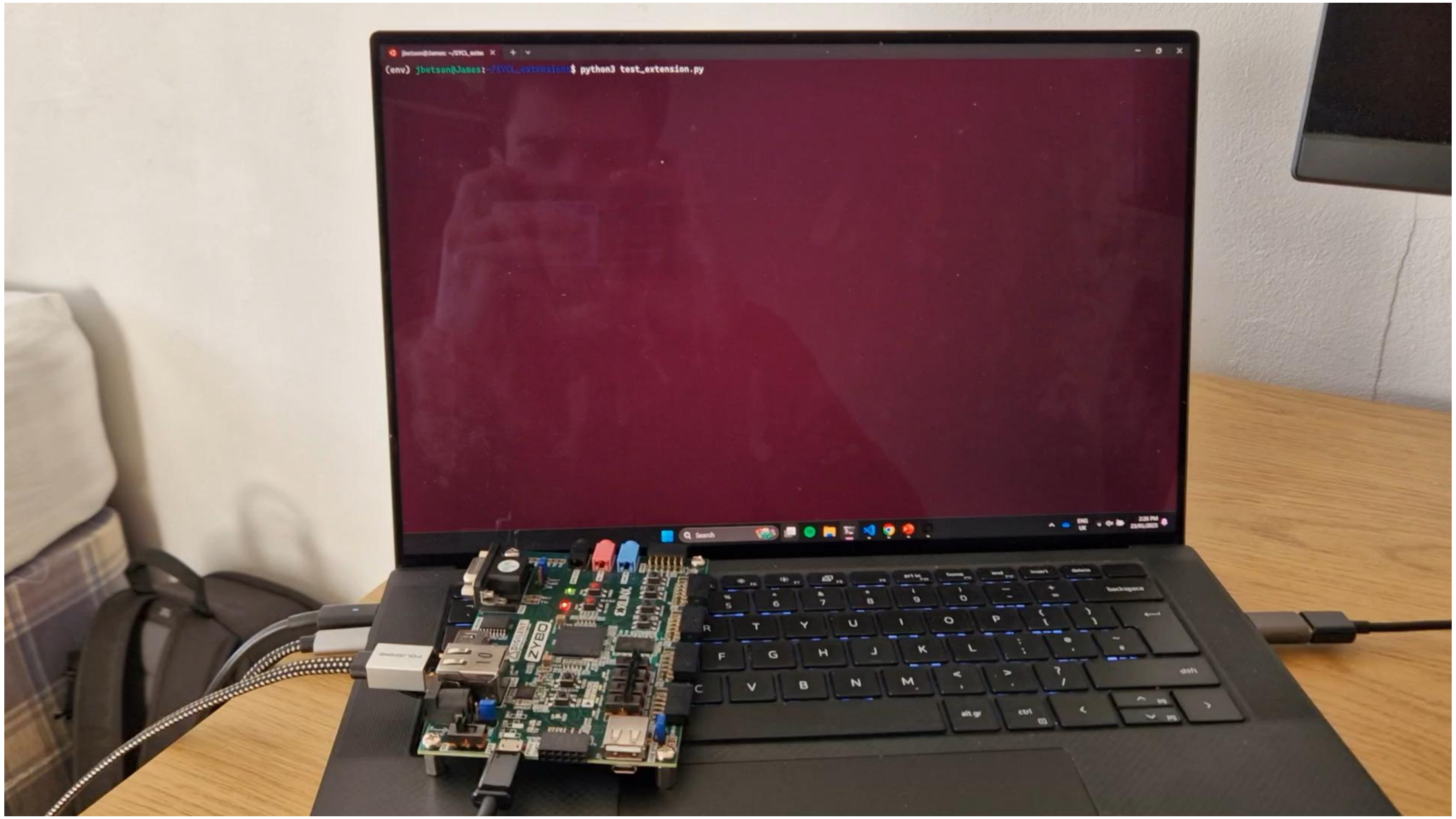
Operation	**Host CPU**	**MicroBlaze V (RISC-V)**	**MicroBlaze**
Add	4.12 µs	23,242.00 µs	25,407.00 µs
Batch_norm	10.87 µs	99,213.64 µs	102,160.08 µs

Conclusion

What we did	**What we didn’t do**
Hardware agnostic PyTorch offload	Accelerate PyTorch
Cheap FPGA hardware solution	Use oneAPI HAL to streamline hardware communication
Investigated OpenCL interposer methodology	

- Expansion to our suite of SYCL PyTorch kernels. Already work underway (SYCL C++ extensions).
- More performant hardware demonstrator. PCIe or CXL solution.
- Investigation into oneAPI HAL.

Video demo



```
(env) jbetson@James:~/SYCL_extension$ python3 test_extension.py
```

GitHub link

Any questions?