# Accelerating numerical libraries for Python using FPGAs

**Gerbrand De Laender**

*Supervisors: prof. dr. ir. Erik D'Hollander & prof. dr. ir. Dirk Stroobandt*

GHENT UNIVERSITY

**Master Thesis – Presentation**

01/09/2021

FACULTY OF ENGINEERING AND ARCHITECTURE

# Overview

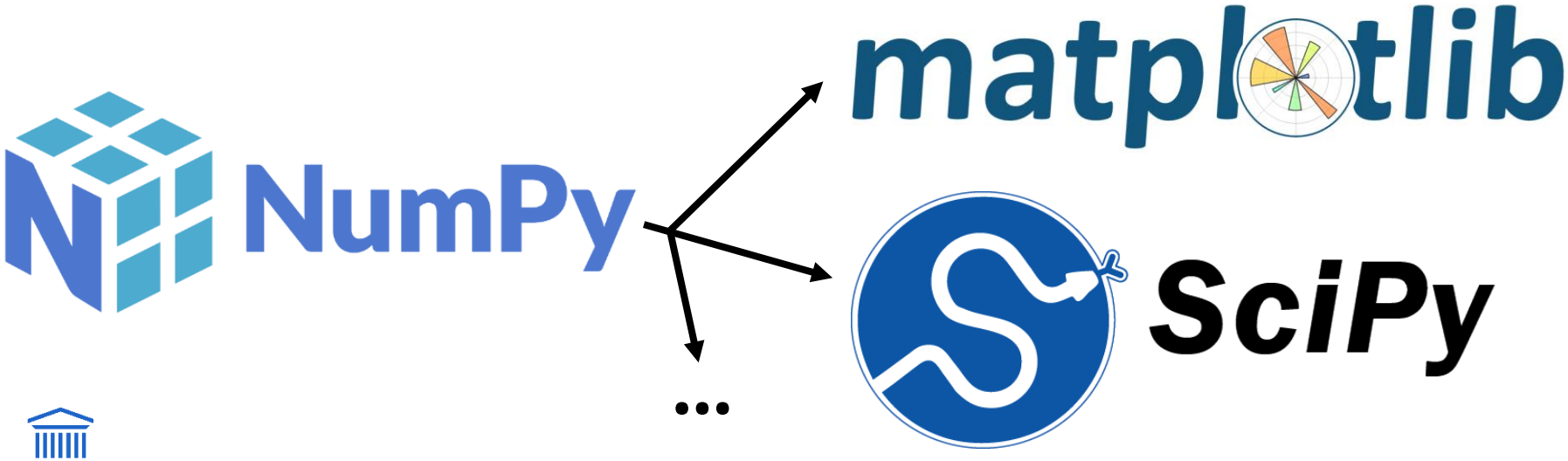
1. Problem statement
2. Thesis goals
3. Methodology & implementation
4. Results
5. Conclusion
6. Questions

# Problem statement

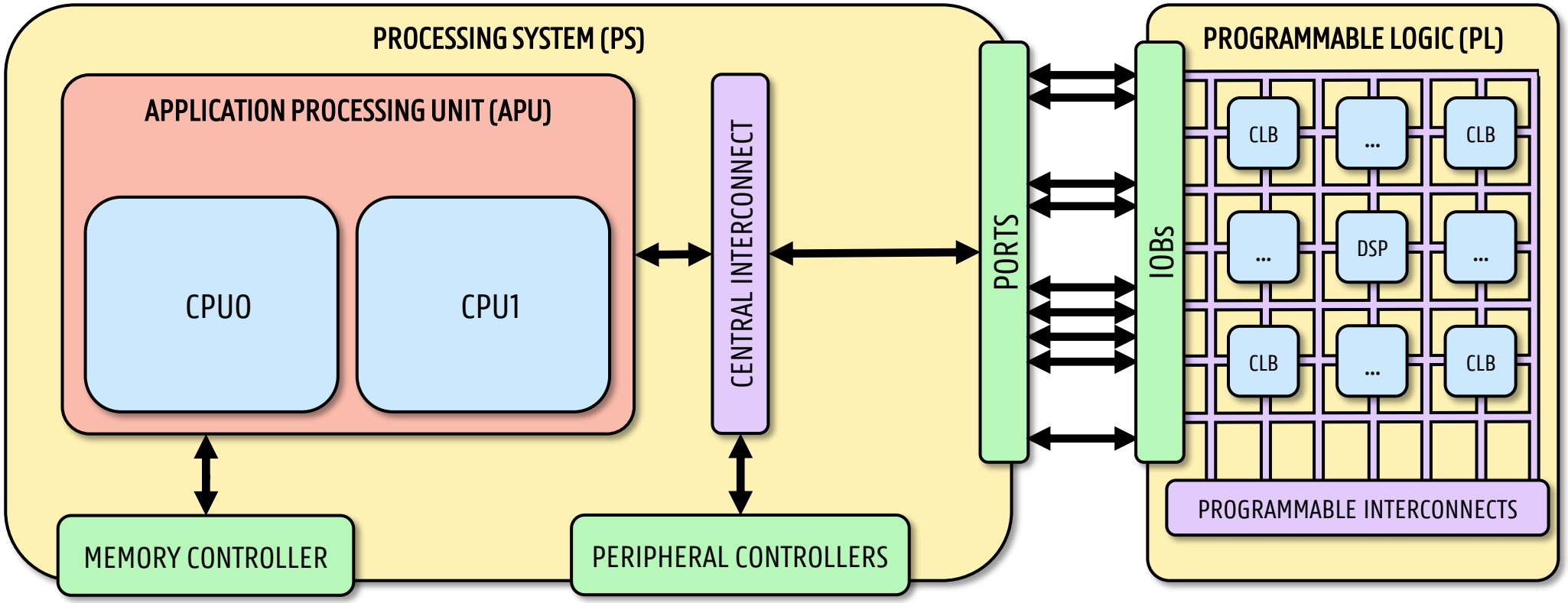
--

# Numerical libraries for Python

- Python is a high productivity language
- Growing popularity for numerical computing
- NumPy is the de-facto standard and has a powerful **ndarray** object

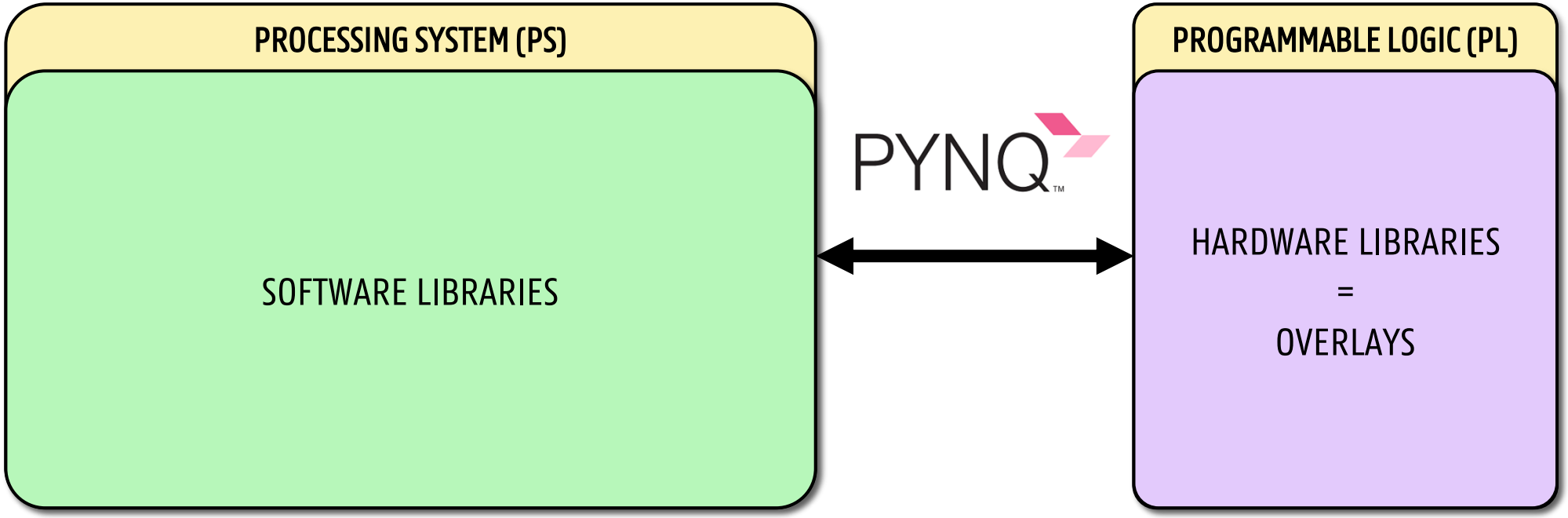


# FPGA acceleration using Zynq-7000 SoC



# Controlling FPGA from Python: PYNQ

- PYNQ presents programmable logic circuits as hardware libraries or *overlays*
- Hardware cores can be controlled using a Python API



# Accelerating numerical libraries for Python using FPGAs

- Many open-source projects… but hardly any that are focussed on NumPy/SciPy
- Many computations accommodate hardware acceleration

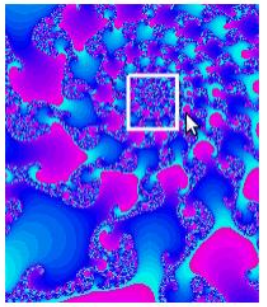
↓

Implement in hardware

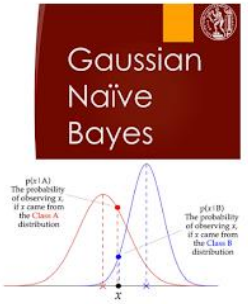
↓

Expected to give an order of magnitude speed-up

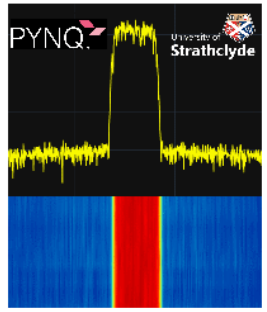
**PYNQ fractal factory**
Fred Kellerman



**Gaussian Naive Bayes**
**NTUA**
Giorgos Tzanos




**Open Source RFSoC Spectrum Analyzer**
University of Strathclyde




**PYNQ ORB feature extractor**
Southeast University, China;
Xilinx China

# Thesis goals

--

# Thesis goals

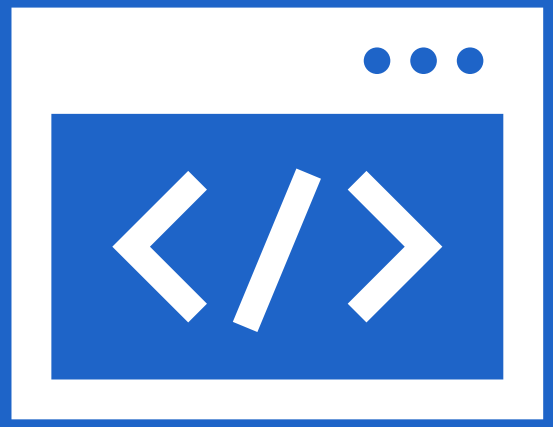
i. **Exploring the interface** between Python and the FPGA.

ii. Implementing selected Python library procedures in hardware using **high-level synthesis.**

iii. Optimising the hardware library routines for **maximum acceleration**.

iv. Creating a mechanism in Python that allows to **transparently invoke a routine** from the hardware library at run-time.

v. **Accelerating a numerical application** in Python.



# Methodology & implementation

--

GHENT UNIVERSITY

Master Thesis – Presentation

01/09/2021

FACULTY OF ENGINEERING AND ARCHITECTURE

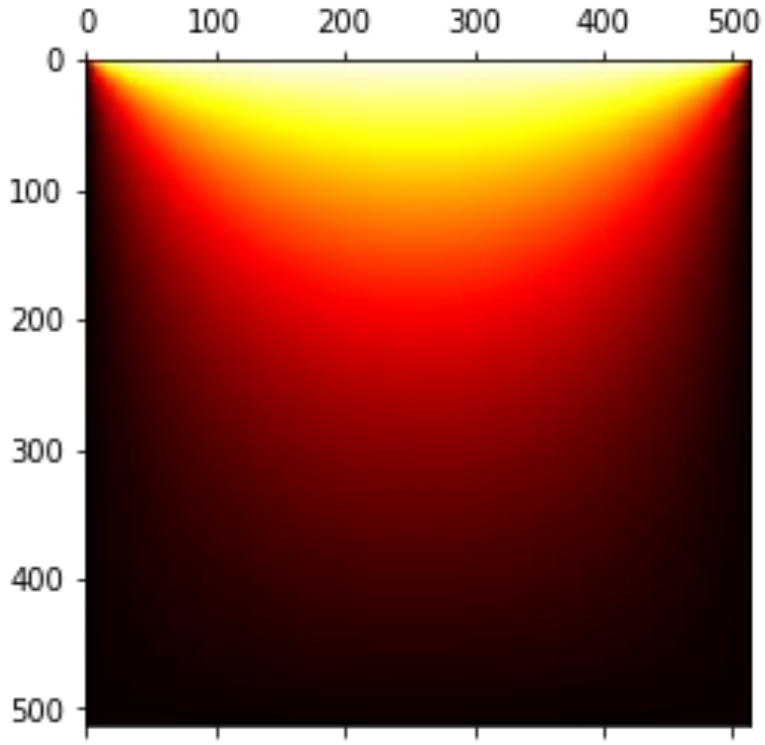
# (1) Identification of hardware procedures

- Which numerical methods are suitable for FPGA acceleration?

Streaming I/O

Low memory footprint

High number of computations



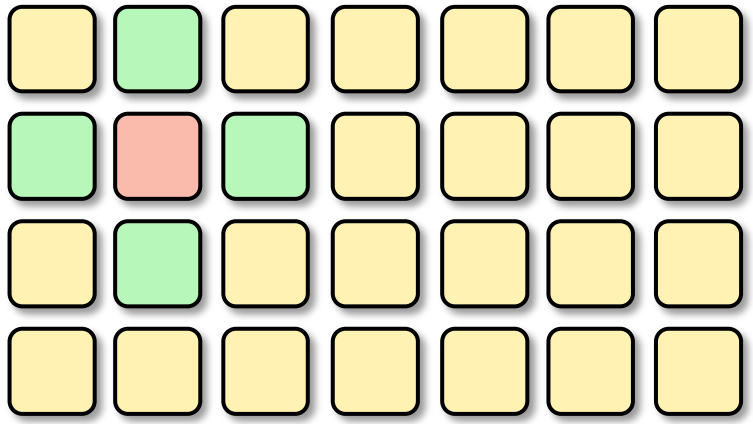
# (1) Identification of hardware procedures

```
# ...
c = grid[1:-1,1:-1]
c_new = np.avg(grid)
delta = np.sum(np.absolute(c - c_new))
c[:] = c_new
```

ELEMENT-WISE OPERATION

REDUCTION-TYPE OPERATION

FILTER-TYPE OPERATION



# (2) Algorithmic description of procedures

- Introduction of streaming behaviour

**Algorithm 1** Processing of stream(s) using an element-wise operation.

1: **function** PROCESSSTREAMS($a[, b], c[, d], length$)
2: $i \leftarrow 0$
3: **while** $i < length$ **do**
4: $c_i[, d_i] \leftarrow f(a_i[, b_i])$ ▷ f is a unary or binary operation
5: $i \leftarrow i + 1$
6: **end while**
7: **end function**

# (3) High-level synthesis implementation

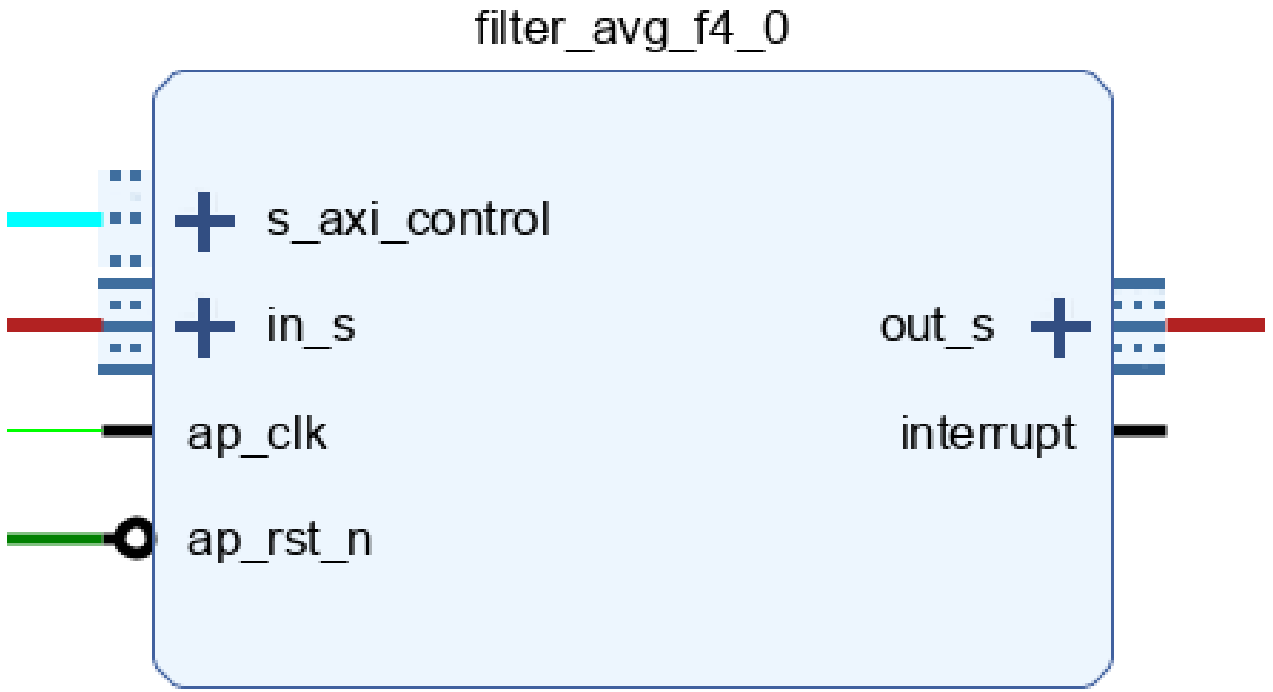
- Using a high-level C/C++ description
- Results in a register transfer level (RTL) design
- Functional validation before and after synthesis

```
1  (...) // Initialisation etc.
2
3  main: do {
4      in1 = read_stream_1(...);   // Read the first input stream.
5      in2 = read_stream_2(...);   // Read the second input stream.
6      out1 = in1 * in2;           // Do the operation.
7      write_stream_1(out1);       // Write to the output stream.
8
9    } while(t_last_not_asserted());
```

# (4) High-level synthesis optimisation

- Feedback provided by synthesis reports
- Optimise for throughput
  - HLS pipeline
  - HLS unroll
  - HLS dependence
  - HLS array_partition
- Optimise for resource utilisation
  - HLS inline
  - HLS allocation
- Other directives
  - HLS interface

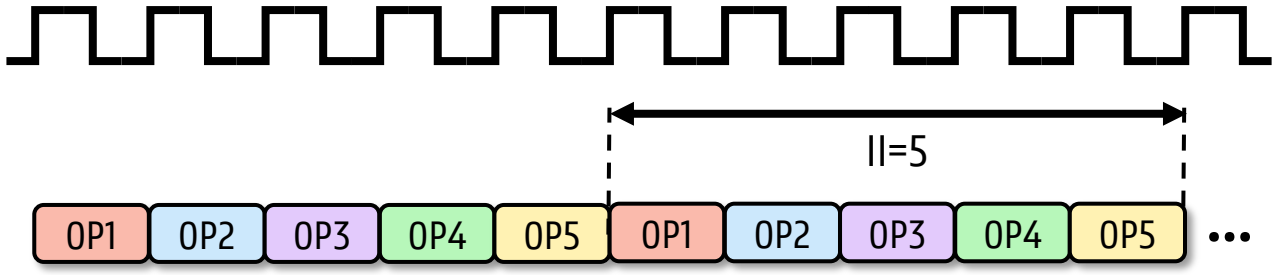
# (4) High-level synthesis optimisation



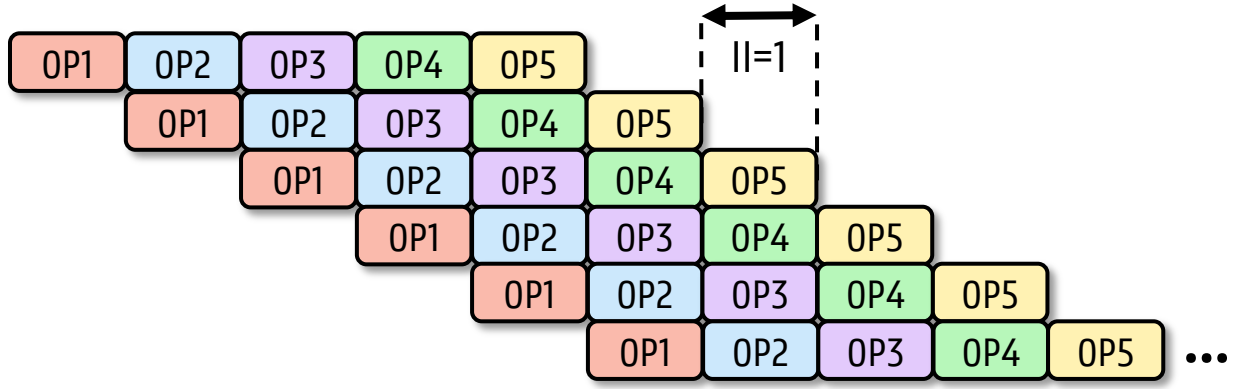

- **HLS interface**
- HLS pipeline
- HLS unroll
- HLS dependence
- HLS array_partition
- HLS inline
- HLS allocation

# (4) High-level synthesis optimisation

```
while(...){
    do_op();
}
```



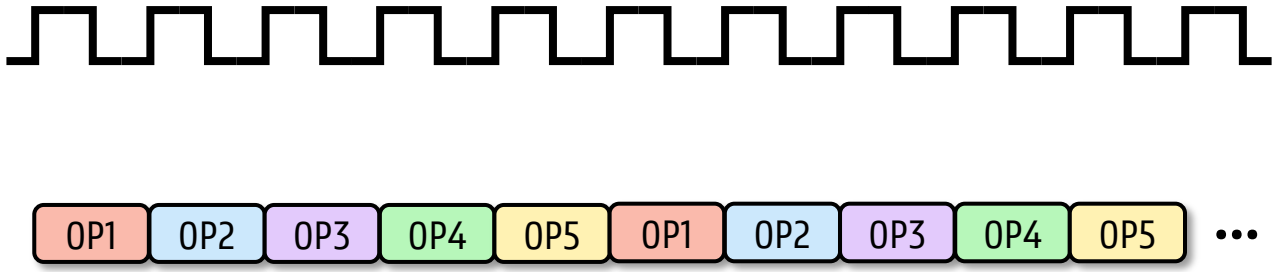

```
while(...){
    #pragma HLS pipeline II=1
    do_op();
}
```



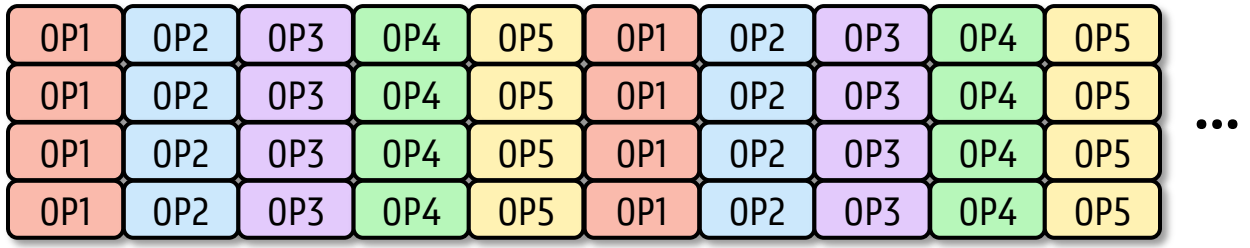

- HLS interface
- **HLS pipeline**
- HLS unroll
- HLS dependence
- HLS array_partition
- HLS inline
- HLS allocation

# (4) High-level synthesis optimisation

```
while(...){
    do_op();
}
```



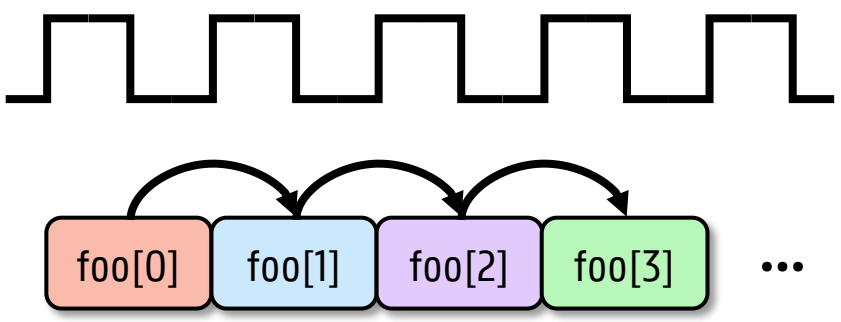

```
while(...){
    #pragma HLS unroll factor=4
    do_op();
}
```



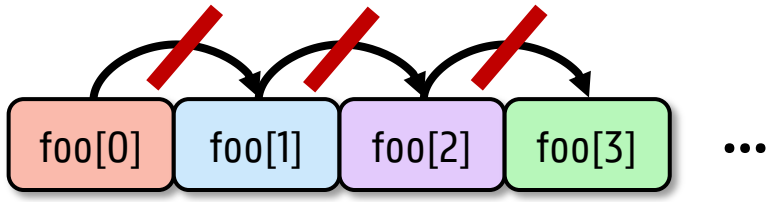

- HLS interface
- HLS pipeline
- **HLS unroll**
- HLS dependence
- HLS array_partition
- HLS inline
- HLS allocation

# (4) High-level synthesis optimisation

```
int i = 0;
while(...){
    foo[i] = bar();
    i = (i + 1) % 4;
}
```



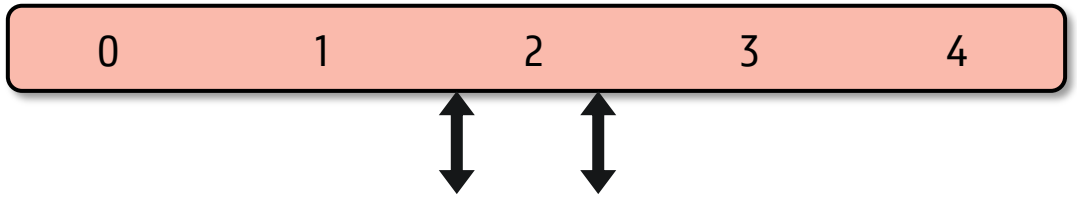

```
int i = 0;
while(...){
    #pragma HLS dependence variable=foo inter false
    foo[i] = bar();
    i = (i + 1) % 4;
}
```



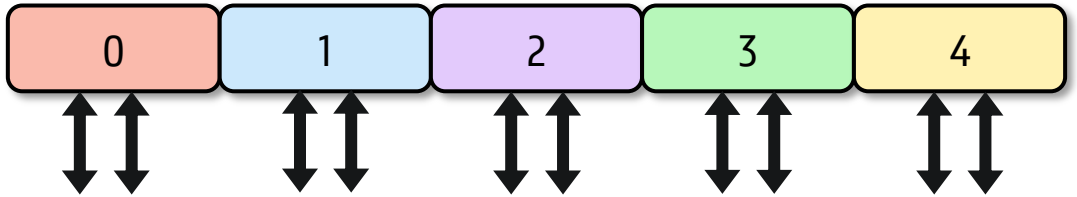

- HLS interface
- HLS pipeline
- HLS unroll
- **HLS dependence**
- HLS array_partition
- HLS inline
- HLS allocation

# (4) High-level synthesis optimisation

```
for(int i = 0; i < 5; i++){
    foo[i] = i;
}
```



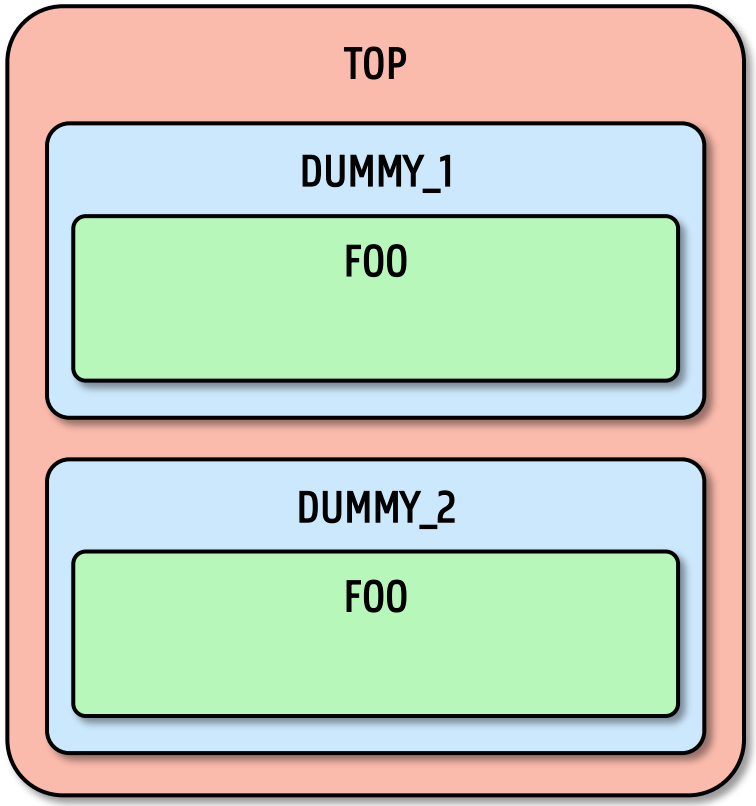

```
for(int i = 0; i < 5; i++){
    #pragma HLS array_partition variable=foo complete
    foo[i] = i;
}
```




- HLS interface
- HLS pipeline
- HLS unroll
- HLS dependence
- **HLS array_partition**
- HLS inline
- HLS allocation

# (4) High-level synthesis optimisation

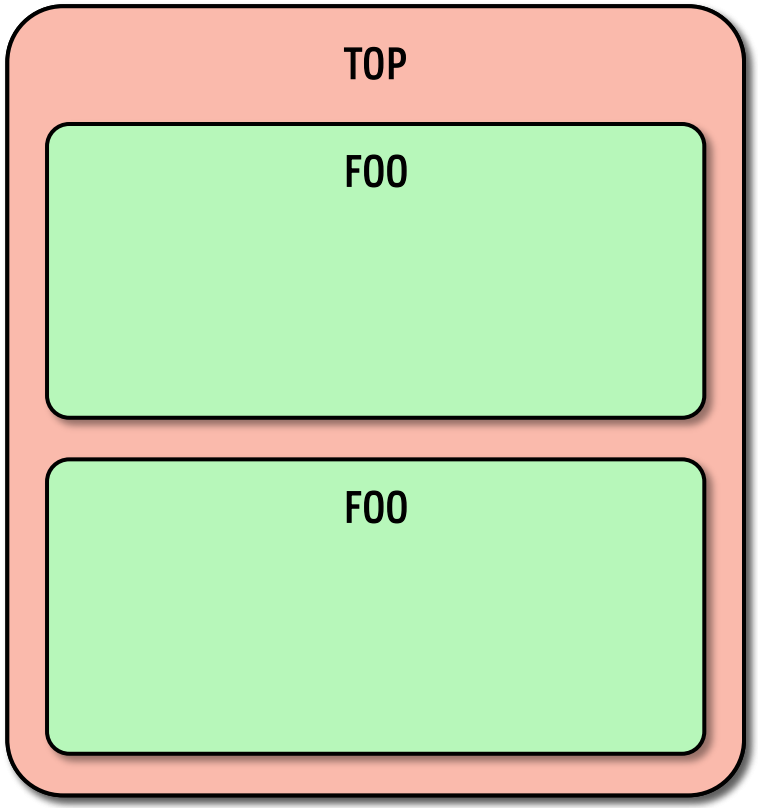
```
void dummy_1(){
    foo();
}

void dummy_2(){
    foo();
}

void top(){
    dummy_1();
    dummy_2();
}
```




- HLS interface
- HLS pipeline
- HLS unroll
- HLS dependence
- HLS array_partition
- **HLS inline**
- HLS allocation

# (4) High-level synthesis optimisation

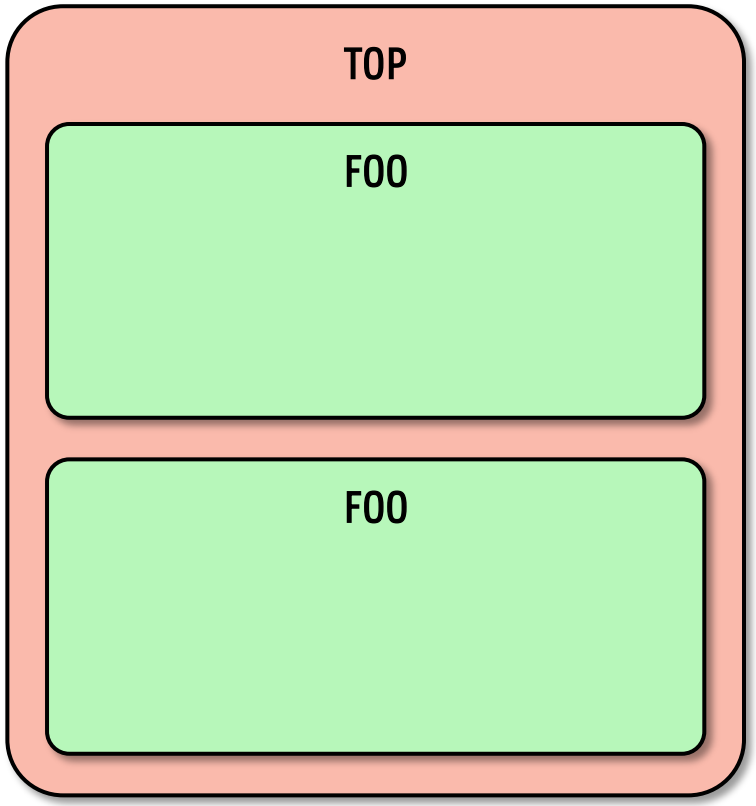
```
void dummy_1(){
    #pragma HLS inline
    foo();
}

void dummy_2(){
    #pragma HLS inline
    foo();
}

void top(){
    dummy_1();
    dummy_2();
}
```




- HLS interface
- HLS pipeline
- HLS unroll
- HLS dependence
- HLS array_partition
- **HLS inline**
- HLS allocation

# (4) High-level synthesis optimisation

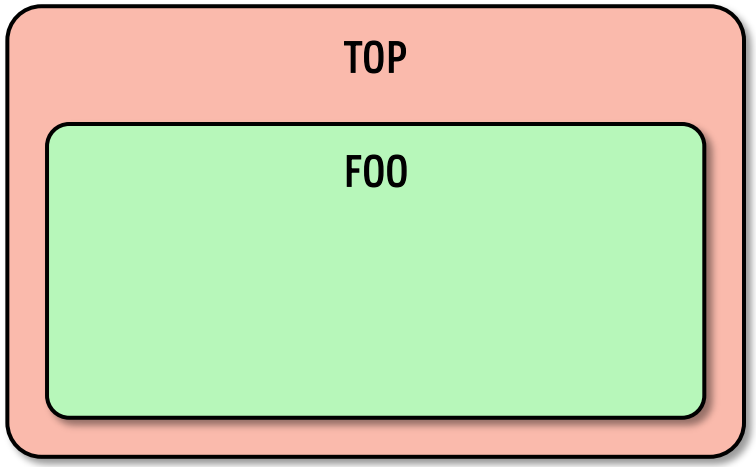
```
void top(){
    foo();
    foo();
}
```




- HLS interface
- HLS pipeline
- HLS unroll
- HLS dependence
- HLS array_partition
- HLS inline
- **HLS allocation**

# (4) High-level synthesis optimisation

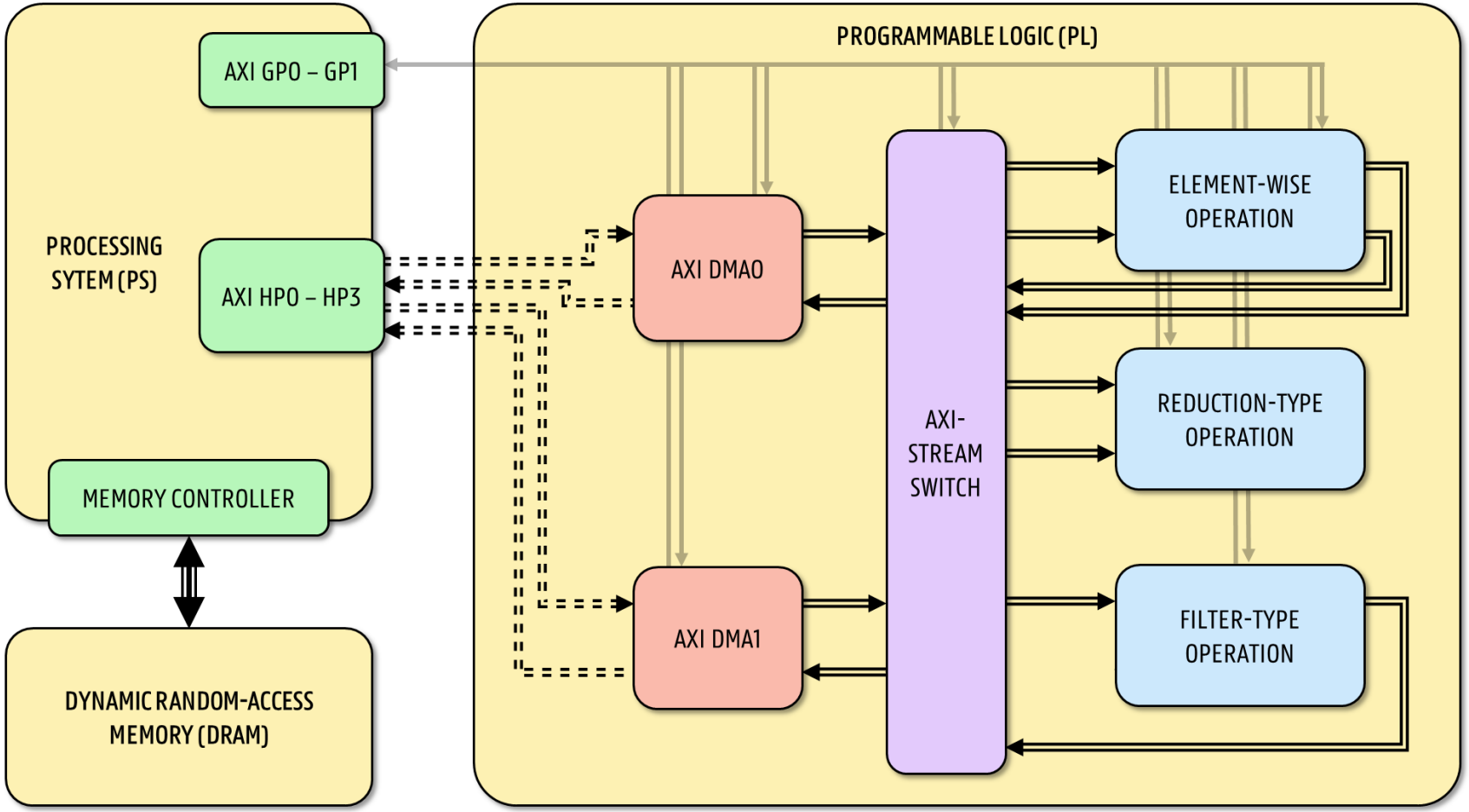
```
void top(){
    foo();
    foo();
}
```




```
void top(){
    #pragma HLS allocation function instances=foo limit=1
    foo();
    foo();
}
```

- HLS interface
- HLS pipeline
- HLS unroll
- HLS dependence
- HLS array_partition
- HLS inline
- **HLS allocation**

# (5) Hardware accelerator integration

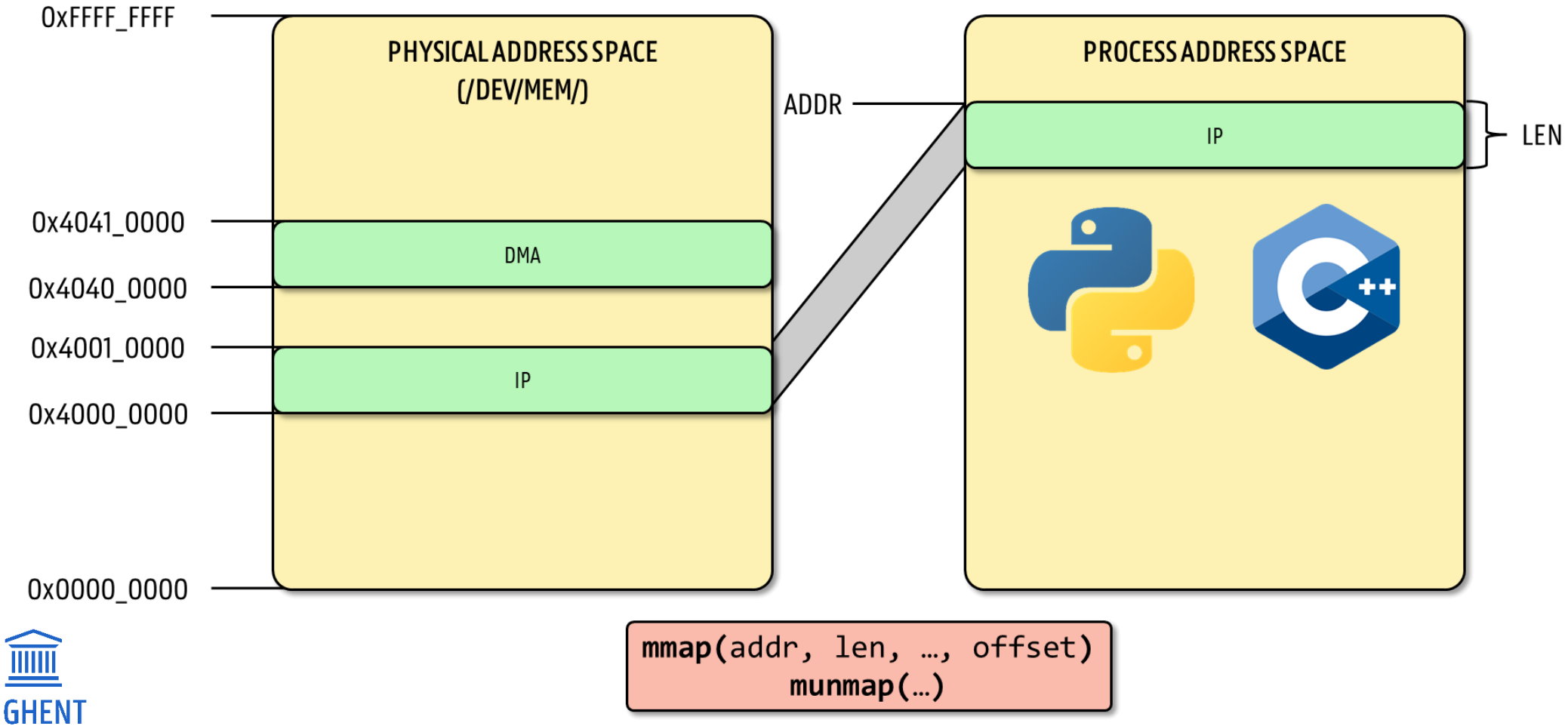


# (6) Creation of lower-level drivers

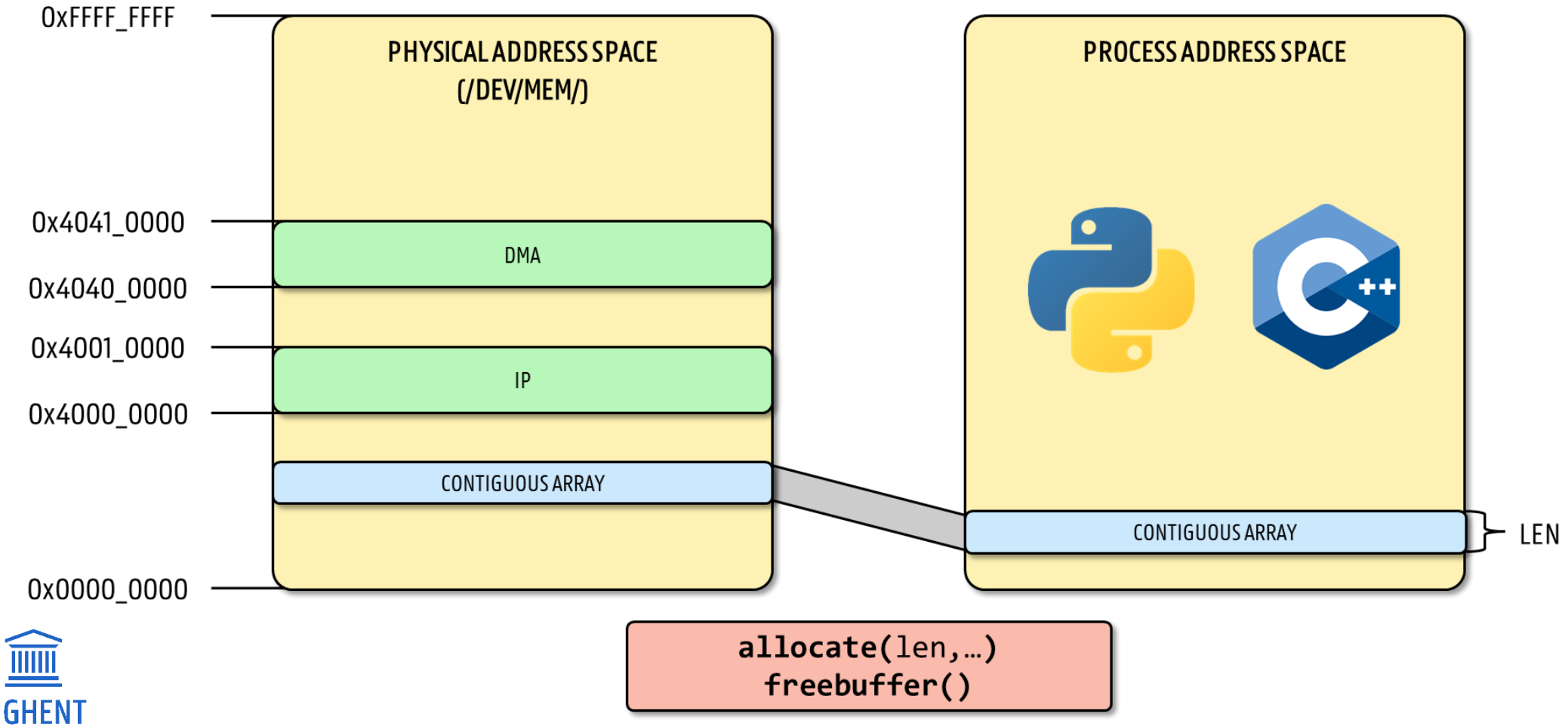
- Use *bare metal* drivers to test hardware design without operating system
- Port these drivers to the Linux operating system

  - FPGA uses physical memory addresses
  - Linux uses virtual memory addresses
  - Take this into account when (i) controlling PL
    (ii) sharing data between PS/PL

# (6) Creation of lower-level drivers



# (6) Creation of lower-level drivers



# (7) Binding of lower-level drivers to Python

- Invoke drivers from a high-level Python application
- Semi-automatic driver generation based on hardware design

CFFI OR pybind11

# (7) Binding of lower-level drivers to Python

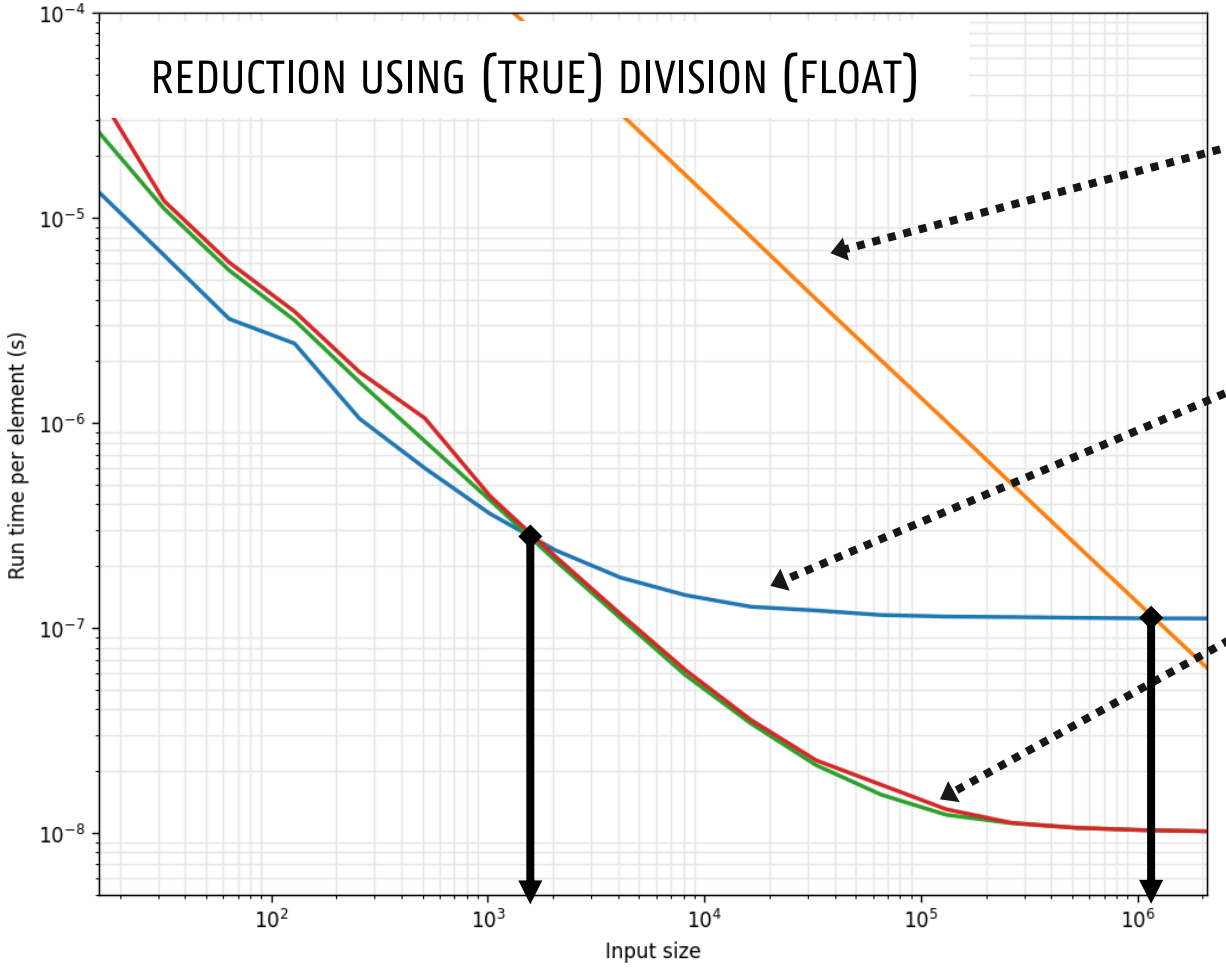
custom_ip_bindings.so

```python
from pynq import allocate, Overlay
from custom_ip_bindings import lib

def custom_ip_python(x1, out):
    assert x1.ndim == out.ndim == 2, "Inputs not supported!"
    lib.custom_ip_c(x1.physical_address, out.physical_address, *x1.shape)

if __name__ == "__main__":
    ol = Overlay("<path to bitstream file>")

    x1 = allocate((64, 64), dtype="f4")
    out = allocate((64, 64), dtype="f4")

    custom_ip_python(x1, out)

    x1.freebuffer()
    out.freebuffer()
```

# (8) Profitability analysis

- Performance gain achieved by hardware accelerator vs. input stream size
- Benchmark to determine crossover point

# (8) Profitability analysis




FPGA accelerated using Python drivers (PYNQ)

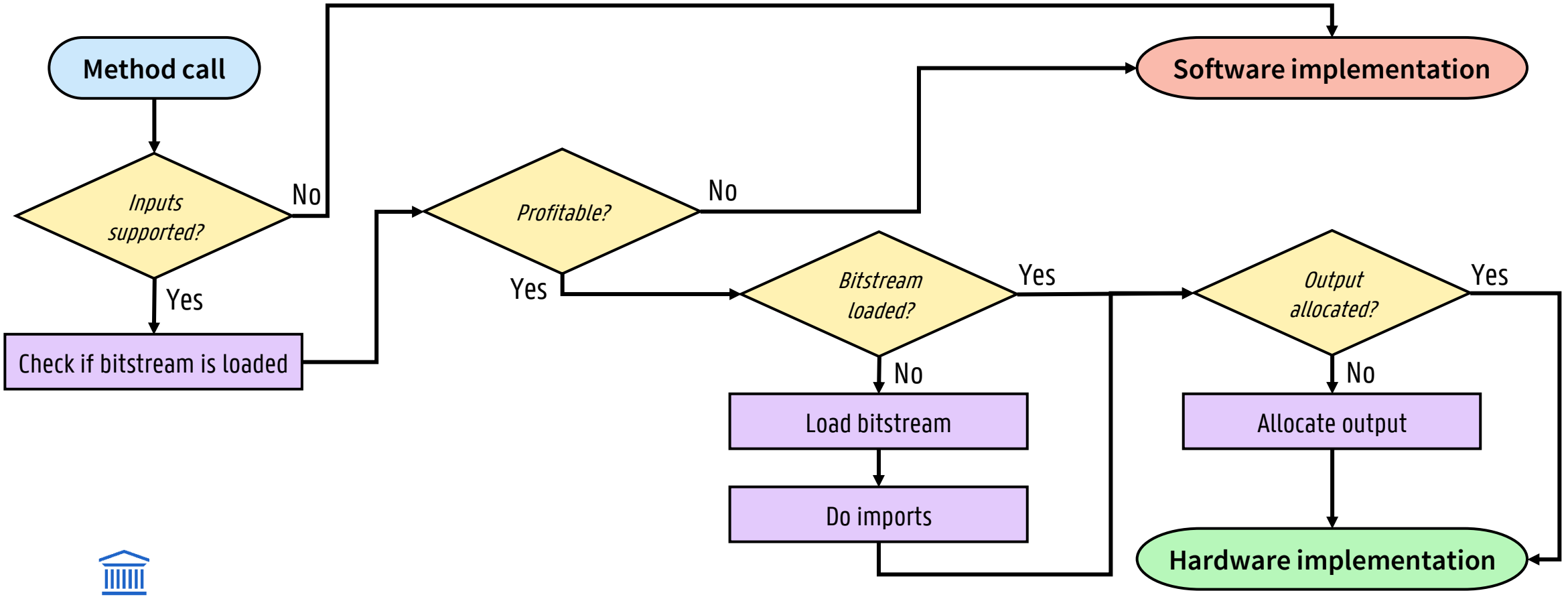
NumPy default (CPU)

FPGA accelerated using C/C++ drivers (CFFI/pybind11)

# (9) Acceleration in Python

- Invoke drivers during run-time of a Python program
- Take into account the available accelerators
- Take into account the profitability analysis
- Take into account the validity of parameters
- **ZyPy** module to provide transparency

# (9) Acceleration in Python



# Results

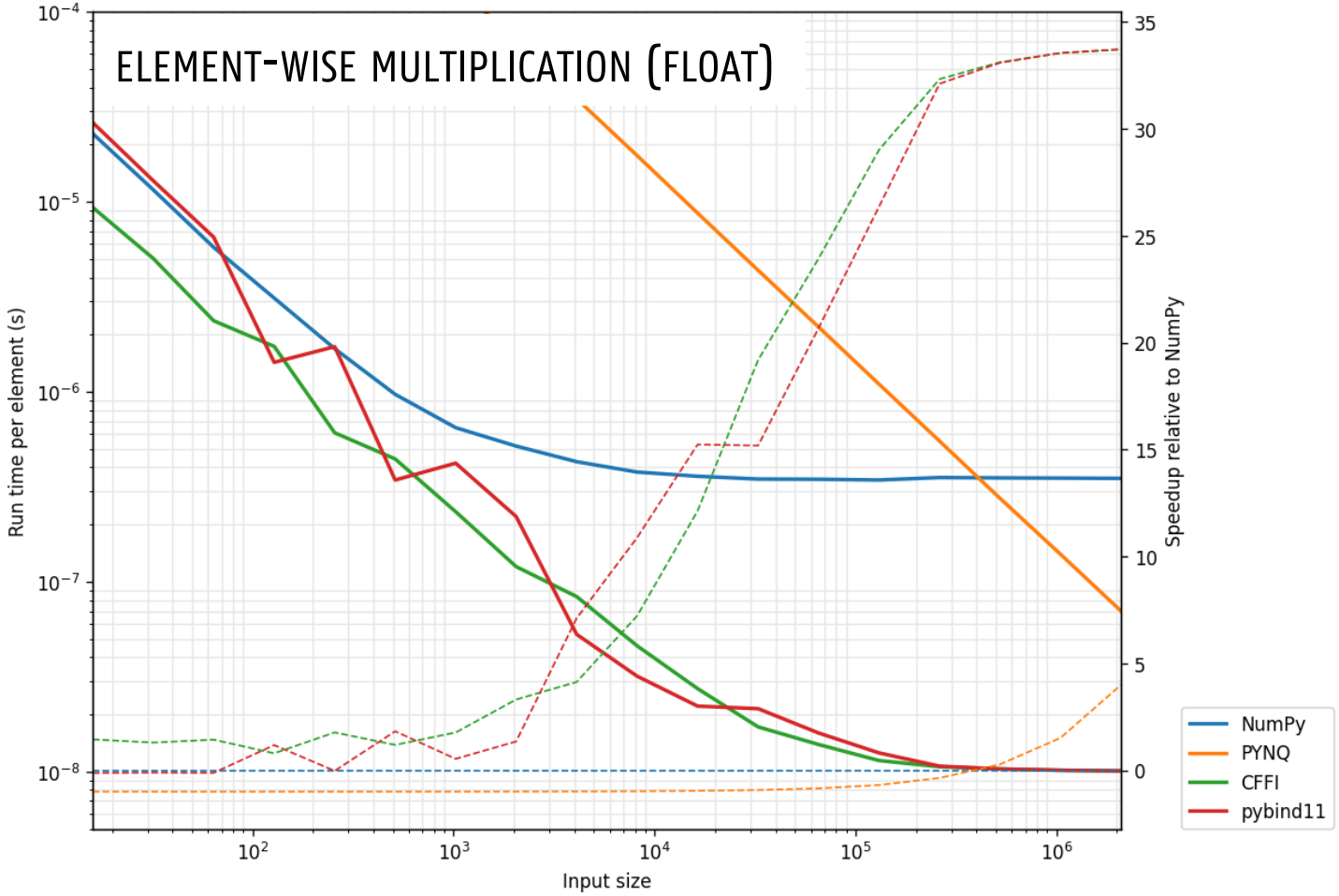
--

# Implemented hardware accelerators

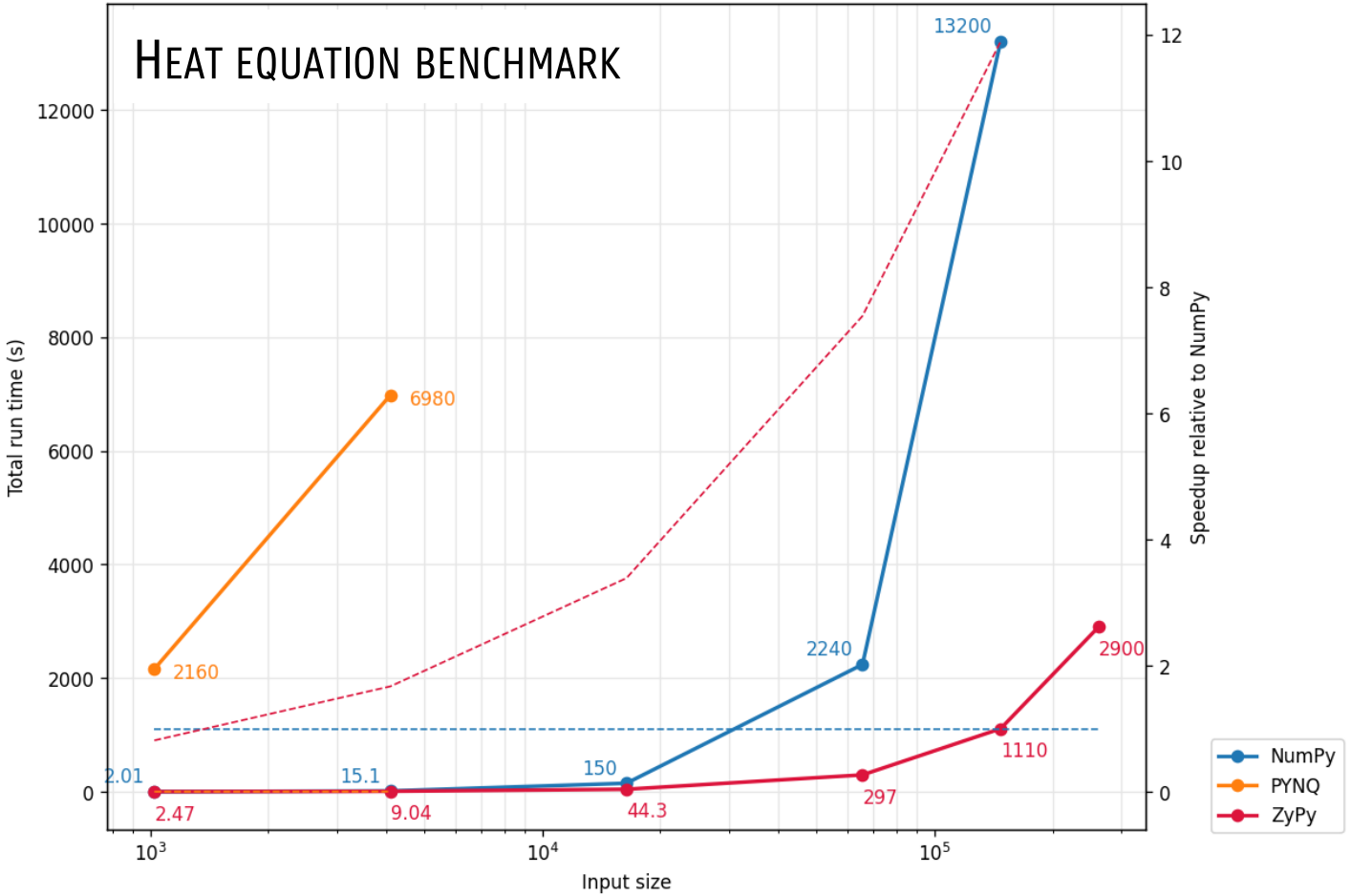
| Accelerator name | Covered NumPy procedures |
|---|---|
| ufunc_call_f4/i4 | f(*) with f = add, subtract, multiply, true_divide, sin, sqrt, arctan, sinh, exp, log |
| ufunc_reduce_all_f4/i4 | f(g.reduce(h(*)) with f = None, square, exp, log<br>g = add, multiply<br>h = None, add, multiply, square, exp, log, absolute(subtract) |
| sad_reduce_all_f4 | sum(absolute(subtract(*))) |
| filter_avg_f4 | 0.2 * (C + N + E + S + W)[1] |

[1] C = x[1:-1, 1:-1], N = x[1:-1, 0:-2], E = x[2:-1,1:-1], S = x[1:-1, 2:-1], W = x[0:-2,1:-1]

# Performance of individual accelerators



# Performance of combination of accelerators



# Conclusion

--

# Conclusion

- Methodology to accelerate numerical libraries in Python using FPGAs
- Custom Python module as drop-in replacement of NumPy
- Integration of multiple hardware accelerators
- Speed-up depends on input size

# Questions

--