EVEREST

DESIGN ENVIRONMENT
FOR EXTREME-SCALE BIG DATA ANALYTICS
ON HETEROGENEOUS PLATFORMS

CPS Summer School – Sep 22, 2022

Tutorial

Olympus: How to use HLS for building customized memory architectures

CHRISTIAN PILATO

EVEREST Scientific Coordinator
Assistant Professor, Politecnico di Milano

christian.pilato@polimi.it

STEPHANIE SOLDAVINI

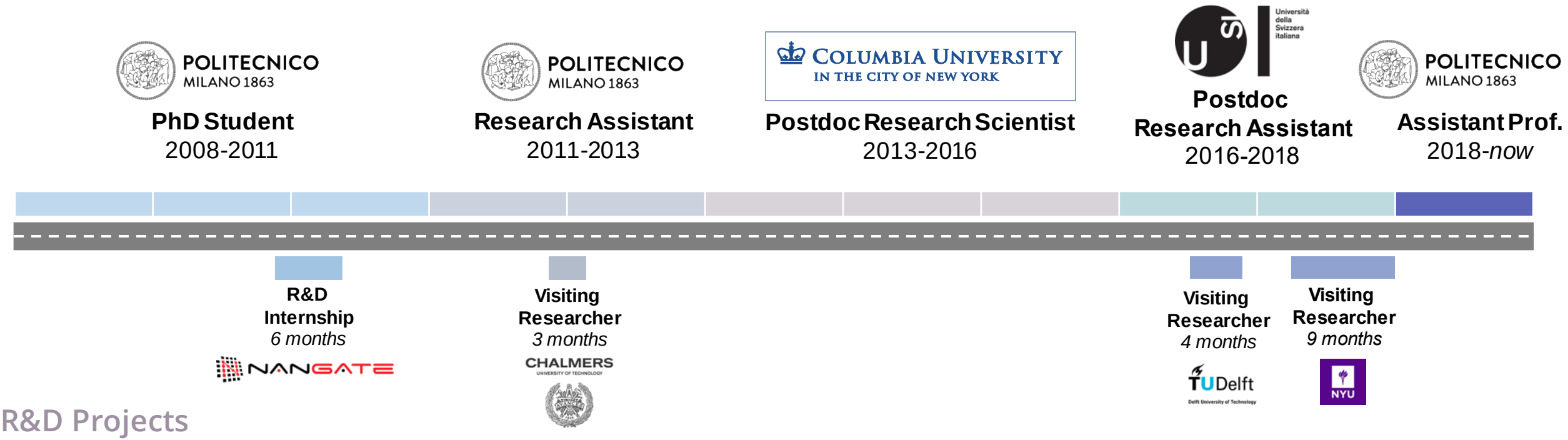
PhD Student, Politecnico di Milano

stephanie.soldavini@polimi.it

About Me – Christian Pilato

Tenure-Track Assistant Professor (RTD)

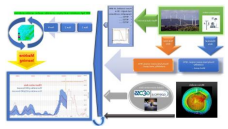
Website: http://pilato.faculty.polimi.it

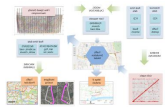


R&D Projects

https://tinyurl.com/CPS2022olympus

The EVEREST Project





Big data applicati
heterogeneous d


- App designers are not FPGA experts
- Hardware accelerators require many optimizations

an have different

H2020 Project – 10 partners, 6 countries

Project Coordinator: Christoph Hagleitner, IBM Research Europe

Scientific Coordinator:

Budget: ~5M€

Start date: October 1, 2020 (36+6 months)

Compilation

Unified hardware gene
(high-level synth

MLIR

Generation of **variants**

Increase quality of accelerators

Improve applications' results

Runtime

adaptation to variants

Virtualization of resources

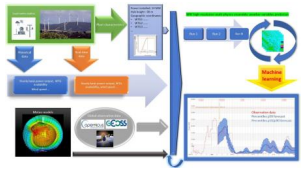
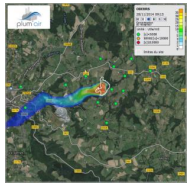
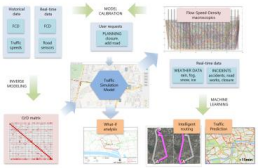
Multi-node support

https://tinyurl.com/CPS2022olympus

EVEREST Approach

Big data applications with heterogeneous data sources

Three use cases

What are the relevant requirements for data, languages and applications?

How to design data-driven policies for computation, communication, and storage?

How to create FPGA accelerators and associated binaries?

How to manage the system at runtime?

How to evaluate the results?

How to disseminate and exploit the results?

EVEREST SDK

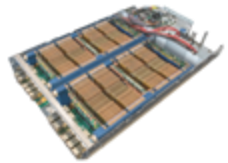
Open-source framework to support the optimization of selected workflow tasks

FPGA-based architectures to accelerate selected kernels

CPU-based infrastructure

+

Two FPGA-based clusters


https://tinyurl.com/CPS2022olympus

EVEREST Partners

IBM Reseach Lab, Zurich (Switzerland)
Project administration, prototype of the target system
PI: Christoph Hagleitner

Università della Svizzera italiana (Switzerland)
Data security requirements and protection techniques
PI: Francesco Regazzoni

Centro Internazionale di Monitoraggio Ambientale (Italy)
Weather prediction models
PI: Antonio Parodi

Virtual Open Systems (France)
Virtualization techniques, runtime extensions to manage heterogeneous resources
PI: Michele Paolino

Numtech (France)
Application for monitoring the air quality of industrial sites
PI: Fabien Brocheton

Politecnico di Milano (Italy)
Scientific coordination, high-level synthesis, flexible memory managers, autotuning
PI: Christian Pilato

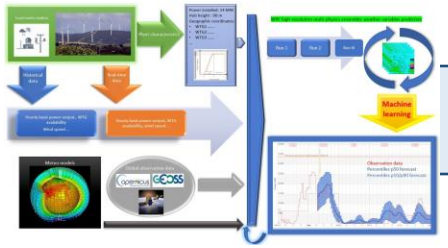
TU Dresden (Germany)
Domain-specific extensions, code optimizations and variants
PI: Jeronimo Castrillon

IT4Innovations (Czech Republic)
Exploitation leaders, large HPC infrastructure, workflow libraries
PI: Katerina Slaninova

Duferco Energia (Italy)
Application for prediction of renewable energies
PI: Lorenzo Pittaluga

Sygic A/S (Slovakia)
Application for intelligent transportation in smart cities
PI: Radim Cmar

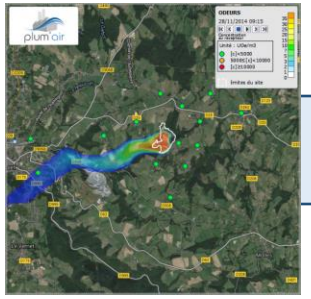
EVEREST Use Cases



Renewable energy production prediction

★ Improve **quality of the predictions**

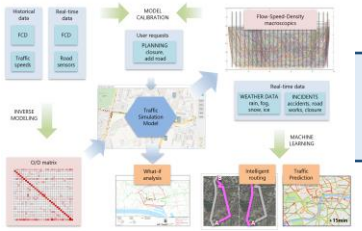
Weather prediction modelling (WRF)



Air-quality monitoring of industrial sites

★ Improve the **response time of predictions**

★ Accelerate kernels to execute more tests



Traffic modeling for intelligent transportation

★ Improve the **overall performance of traffic simulation**

Accelerated computationally-intensive kernels + **Machine-learning** kernels

https://tinyurl.com/CPS2022olympus

The Case of Computational Fluid Dynamics

Numerical simulations are becoming more and more popular for many applications

- **Computational Fluid Dynamics** requires to solve partial differential equations
- **Inverse Helmholtz operator** ("Helmholtz" for the friends) is parametric with respect to polynomial degree *p*

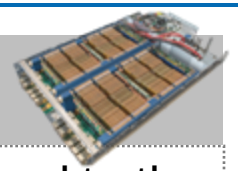
```
var input S    : [11 11]
var input D    : [11 11 11]
var input u    : [11 11 11]
var output v   : [11 11 11]
var t          : [11 11 11]
var r          : [11 11 11]
t = S # S # S # u . [[1 6] [3 7] [5 8]]
r = D * t
v = S # S # S # t . [[0 6] [2 7] [4 8]]
```

Final result is obtained by **"small" contributions** on **independent data**

- CFD kernel is composed of **three high-level tensor operators** (two contractions and one Hadamard product) repeated millions of times – good for spatial parallelism
- Each operator requires $p^2 + 2 \cdot p^3$ *(double) elements* as input and produces p^3 *(double) elements* – 21.74 KB + 10.40 KB per element when p = 11
- *Six tensors (p^3 elements)* to store intermediate results – additional 62.39 KB

EVEREST Target System

cloudFPGA



- **Disaggregated FPGAs** directly attached to the network (64 FPGA instances)
- **Low latency** and **high bandwidth** system
- Separation between **Shell** and **Role** modules
- **cFDK framework** for system generation

FPGA-Accelerated HPC Cluster

- Cluster of **PCIe-attached FPGAs** (Alveo) with HBM architecture (up to 460 GB/s per board)
- **Xilinx Vitis framework** for HLS and system integration
- Support for the integration of **custom HDL**

CPU Reference System

- CPU-based infrastructure to **execute end-to-end workflows**, **manage storage**, and **data transfers**
- Extended to support the **offloading of tasks** to **FPGA servers**

LEXIS

Exploit **spatial parallelism**

High **memory bandwidth**

Different nodes to better **match applications**

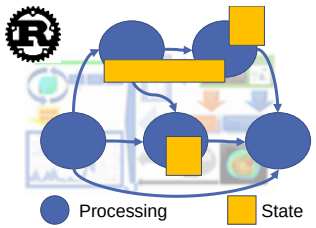
Data-intensive (memory-bound) applications

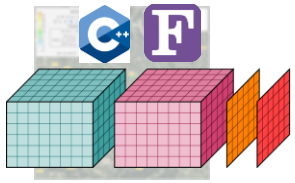
Seamless support for multiple nodes

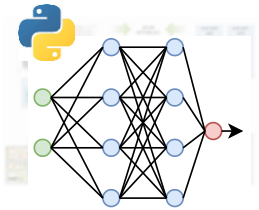
Limited **FPGA resources** (esp. memories)

https://tinyurl.com/CPS2022olympus

EVEREST System Development Kit (SDK)







Different **input flows**
starting from different **input languages**

Support for **multiple target boards**

Collection of interoperable and open-source tools to create hardware/software systems that can adapt to the target system, the application workflow, and the data characteristics

- Compilation framework based on **MLIR** to unify the input languages
- **High-level synthesis** and **hardware generation flow** to automatically create optimized architectures
- Creation of hardware and software variants to match architecture features
- Use of state-of-the-art frameworks and commercial toolchains for FPGA synthesis

tvm
MLIR
DandA
XILINX VITIS
cFDK
HEAppE
HyperLoom
(and more...)

https://tinyurl.com/CPS2022olympus

Challenges for HPC Architectures (i)

Challenge 1: Input languages and frameworks

- Application designers are usually not FPGA experts and may use high-level framework that are not supported by current HLS tools – how to talk with them?

Challenge 2: CPU-Host Communication Cost

- FPGA logic requires the data on the board, but data transfers can be much more expensive than kernel computation – can we execute more than one point?

Challenge 3: Read/Write Burst Transactions

- We need to determine the proper size of the transactions to get the maximum performance – how to reorganize the data transfers and get the parameters?

Challenges for HPC Architectures (ii)

Challenge 4: Full Bandwidth Utilization

- AXI interfaces may be large (e.g., 256 bits on the Alveo) – how to leverage them?
- HBM architectures have many channels - how to parallelize data transfers?

Challenge 5: Data Allocation

- Data must be placed in memory to maximize its utilization but also to enable efficient data transfers/computation – custom data layouts?

Challenge 6: Synthesis-Related Issues

- FPGA devices are large but still not sufficient for hosting many kernels – how to trade-off optimizations and parallel instances?
- FPGA architectures may be different – separate platform agnostic and dependent parts?
- FPGA logic architectures are complex and may introduce performance degradation – how to "guide" the synthesis process?

Hardware HPC (Memory) Architectures

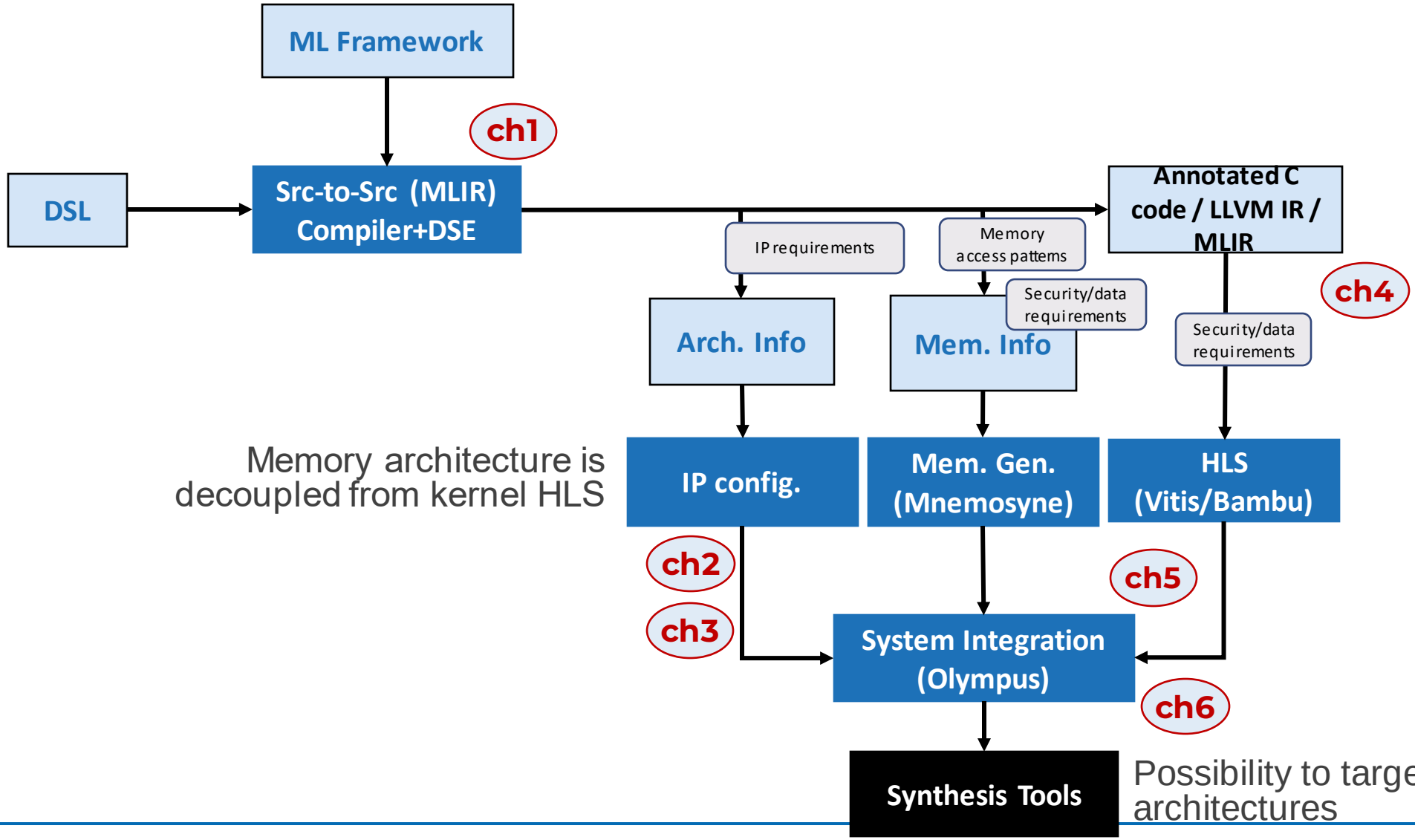
What do we mean with **memory architecture**?

Every hardware module that is responsible to provide data to the accelerator kernels

Additional issues:

- BRAM resources are limited
  - Helmholtz operator requires >94 KB of local data
  - If local storage is not optimized, the number of parallel kernels can be limited
- Application-specific details can be used to optimize the data transfers
  - In Helmholtz, one of the tensors is constant over all elements – how to match these details with platform characteristics?
  - Better to transfer data for a "batch" of elements and then execute them in series – how many? again, limited storage

Hardware Compilation Flow



Memory architecture is decoupled from kernel HLS

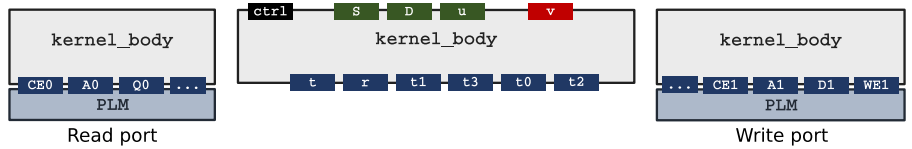
Possibility to target different architectures

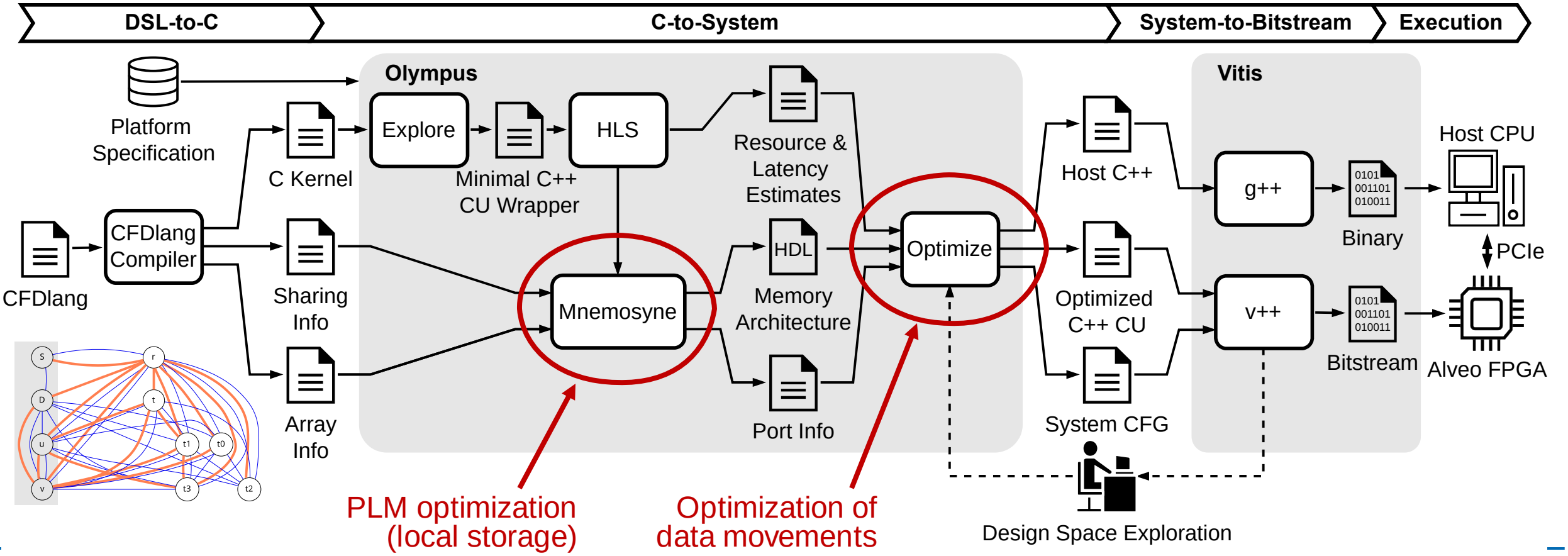
https://tinyurl.com/CPS2022olympus

From DSL to Bitstream – Focus on Memory

```
var input S    : [11 11]
var input D    : [11 11 11]
var input u    : [11 11 11]
var output v   : [11 11 11]
var t          : [11 11 11]
var r          : [11 11 11]
t = S # S # S # u . [[1 6] [3 7] [5 8]]
r = D * t
v = S # S # S # t . [[0 6] [2 7] [4 8]]
```

```
void kernel_body(double S[11][11], double D[11][11][11], double u[11][11][11],
                 double v[11][11][11],
                 double t[11][11][11], double r[11][11][11], double t1[11][11][11],
                 double t3[11][11][11], double t0[11][11][11], double t2[11][11][11])
```





https://tinyurl.com/CPS2022olympus

EVEREST

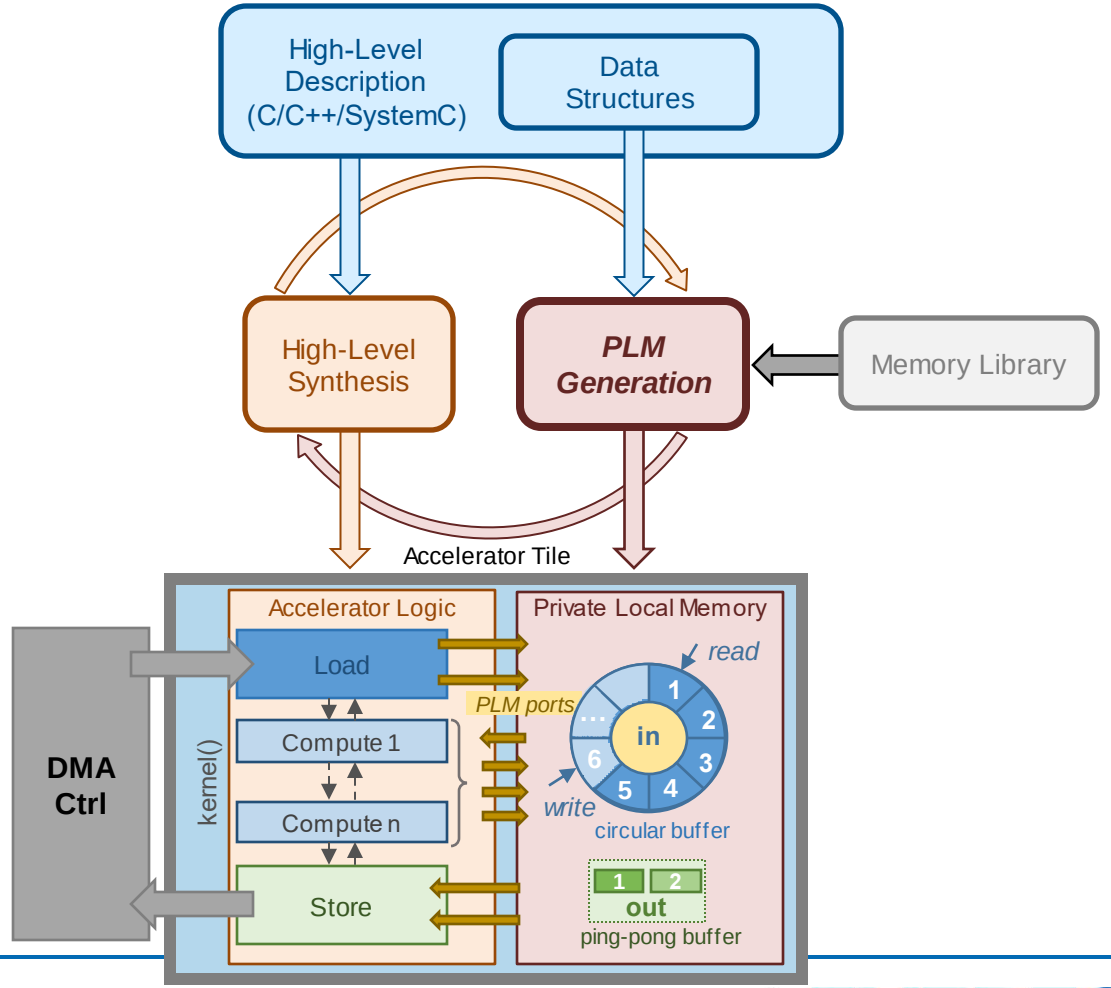
PLM Customization for Heterogeneous SoCs

High-Level Synthesis (HLS) to create the accelerator logic

- Definition of memory-related parameters (e.g., number of process interfaces)

Generation of **specialized PLMs**

- Technology-related optimizations
- Possibility of system-level optimizations across accelerators



https://tinyurl.com/CPS2022olympus

EVEREST

Reuse What is not Used

Generally, we use one **PLM unit** (possibly composed of many banks) for each **data structure** (array)

Reuse the same memory IPs
for several data structures

"Two data structures are compatible if they can be allocated to the same PLM unit (memory IPs)"

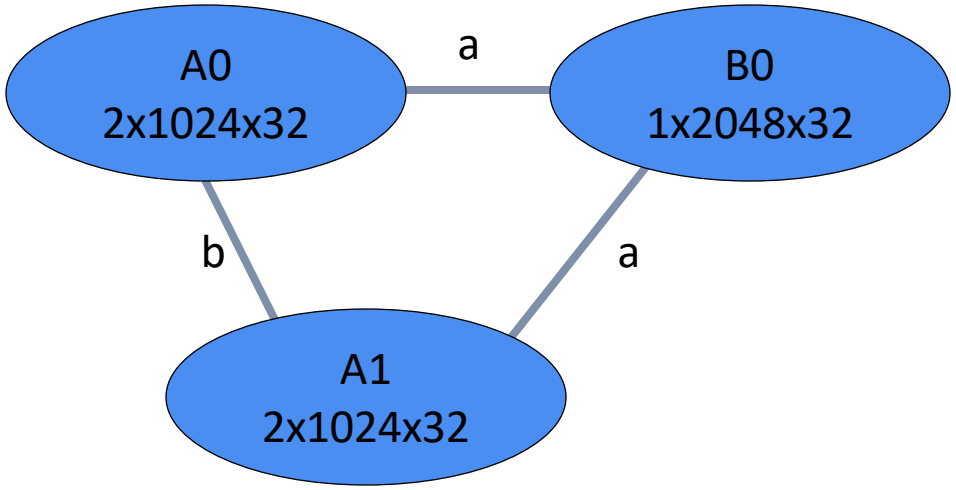
A common case: accelerator kernels never executed at the same time

- Possible only at system-level, when integrating the components
- Optimizations of accelerator logic and memory subsystem are independent

Memory Compatibility Graph (MCG)

Graph to represent the possibilities for optimizing the data structures

- Each node represents a data structure to be allocated, annotated with its data footprint (after data allocation)
- Each edge represents compatibility between the two data structures
- Can be automatically extracted from the MLIR-based compiler flow
  - Variant exploration to achieve the "best solutions"

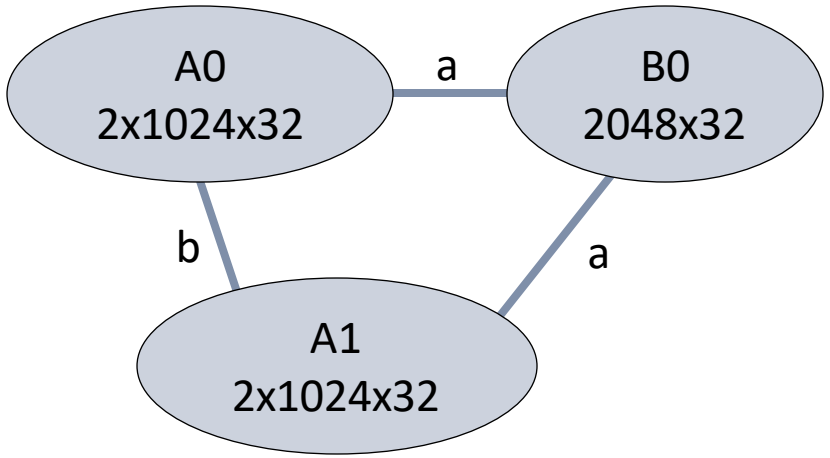


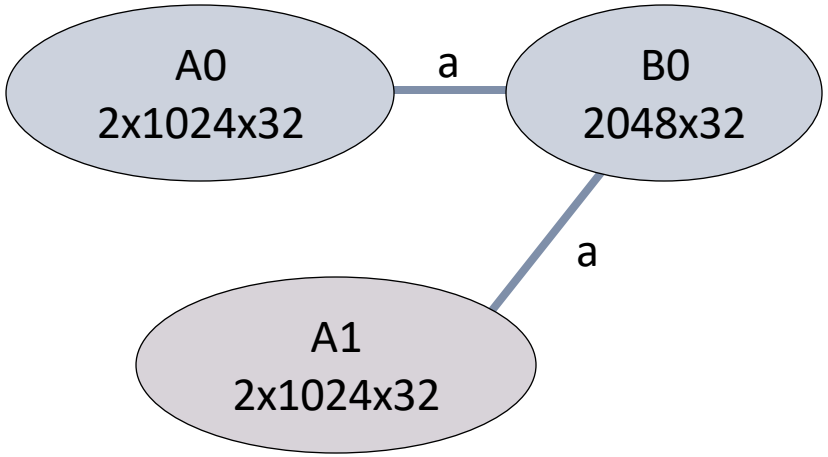
a) **Address-space compatibility**: the data structures are compatible and can use the same memory IPs
b) **Memory-interface compatibility**: the ports are never accessed at the same time and the data structures can stay in the same memory IP

Clique Definition

“A clique is a subset of the vertices of the memory compatibility graph such that every two vertices are connected by an edge”

A clique represents a **set of data structures** that can share the same memory IPs



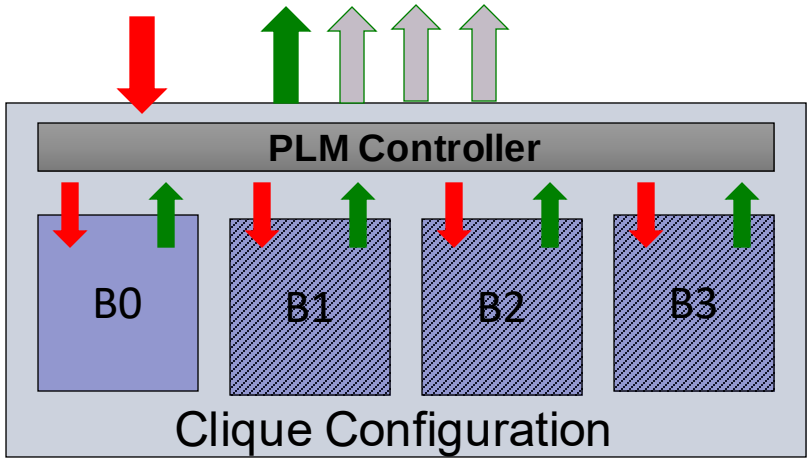


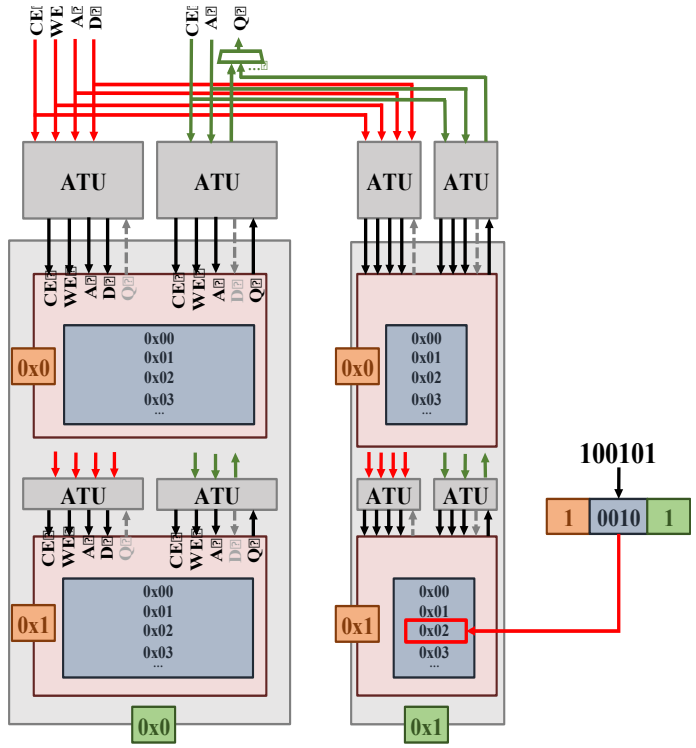
We need two distinct configurations!
{A0,B0} and {A1} or {A1,B0} and {A0}?

PLM Controller Generation

A **lightweight PLM controller** is created for each compatibility set (clique) based on the bank configuration

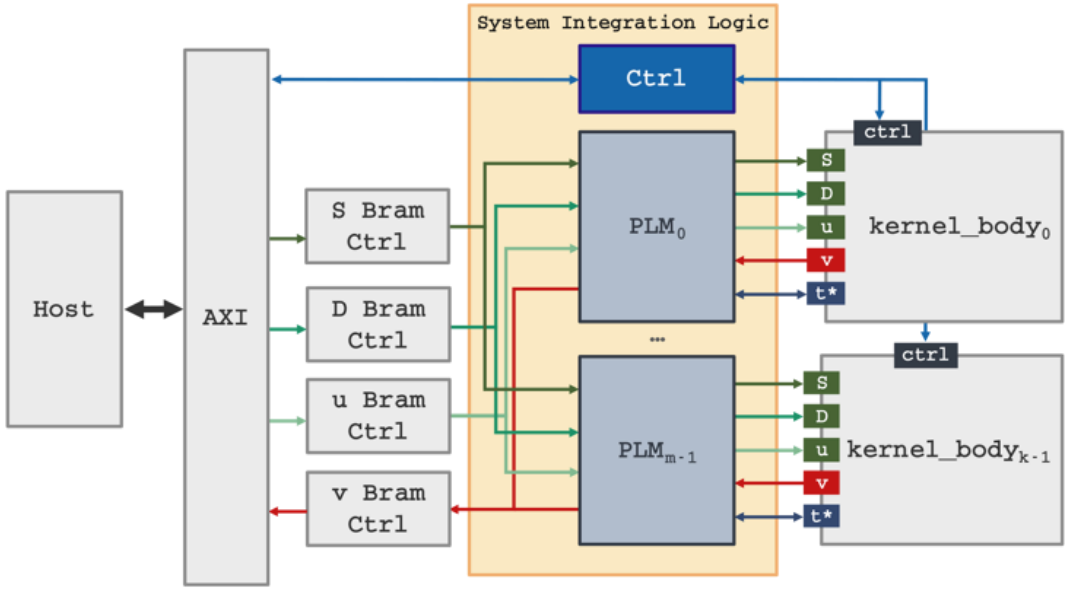
- Accelerator logic is not aware of the actual memory organization
- Array offsets need to be translated into proper memory addresses

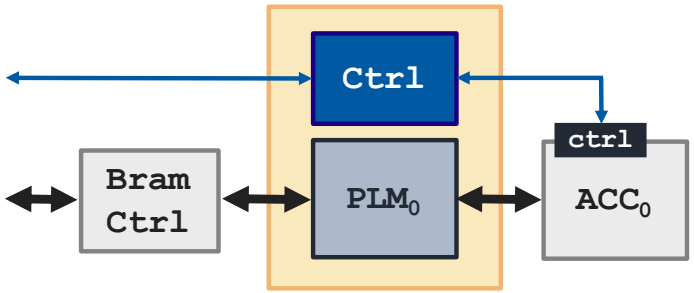


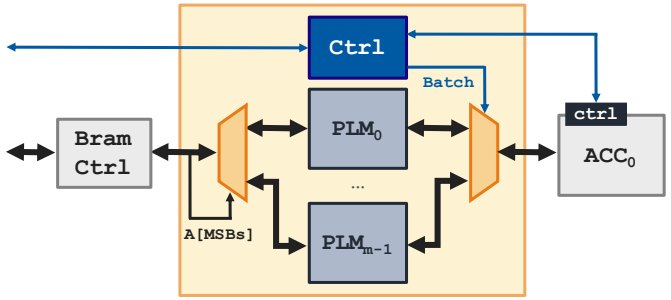


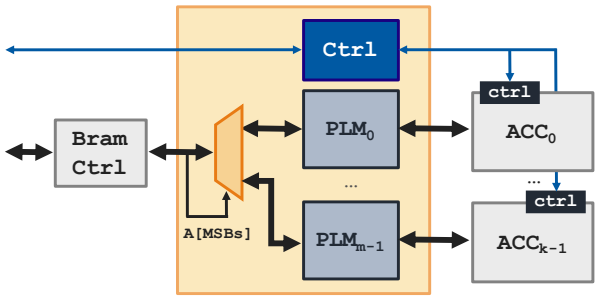
Custom logic with negligible overhead, especially when the number of banks and their size is a power of two

Creation of Parallel Architectures









K. F. A. Friebel, S. Soldavini, G. Hempel, C. Pilato, J. Castrillon. "From Domain-Specific Languages to Memory-Optimized Accelerators for Fluid Dynamics" HPCFPGA'21

https://tinyurl.com/CPS2022olympus

Olympus – Automated System-Level Integration

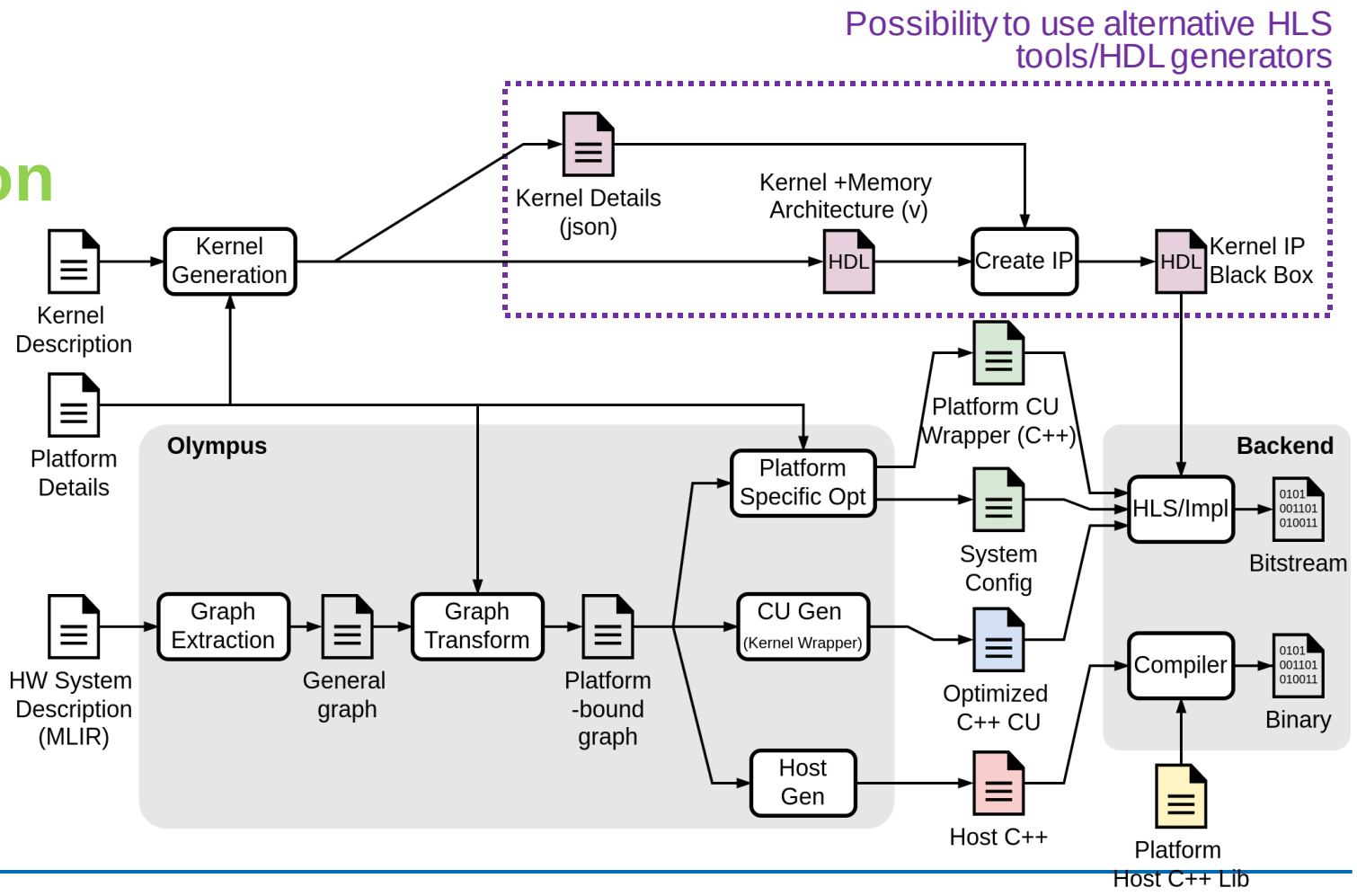
Complete **hardware architecture generation flow** from high-level specification

Platform-specific description

- HBM-based Xilinx Alveo
- IBM CloudFPGA
- …

Host code generation

- Based on **platform libraries** for the specific target



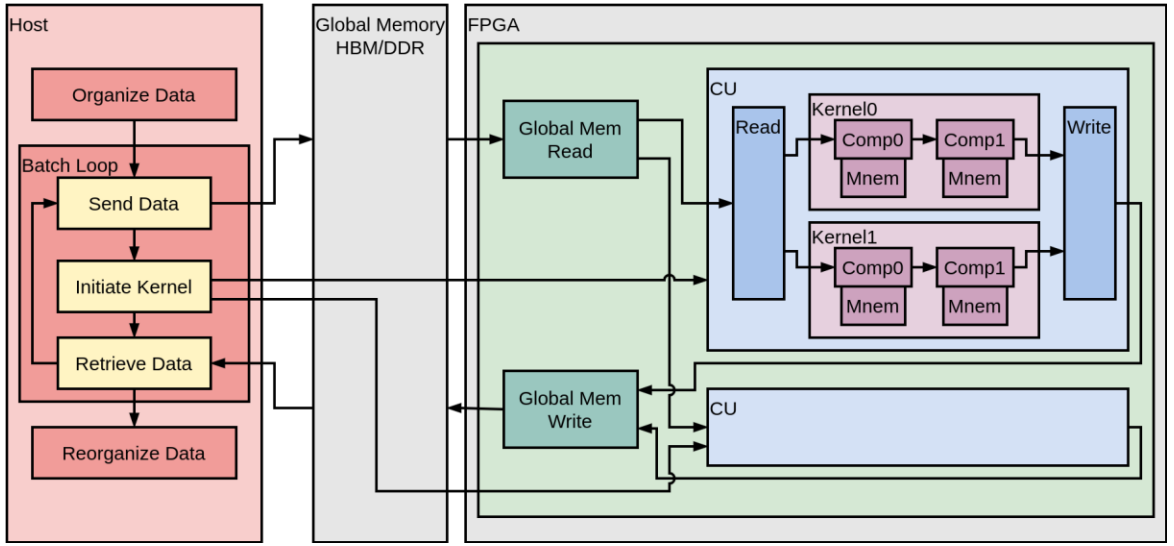
https://tinyurl.com/CPS2022olympus

EVEREST

Olympus – System generation flow

Determines the **system-level architectures** based on:

- **Algorithm parallelism**
- Characteristics of the **target platform(s)**
- Interfaces of the modules (**HLS tools**)



Produces

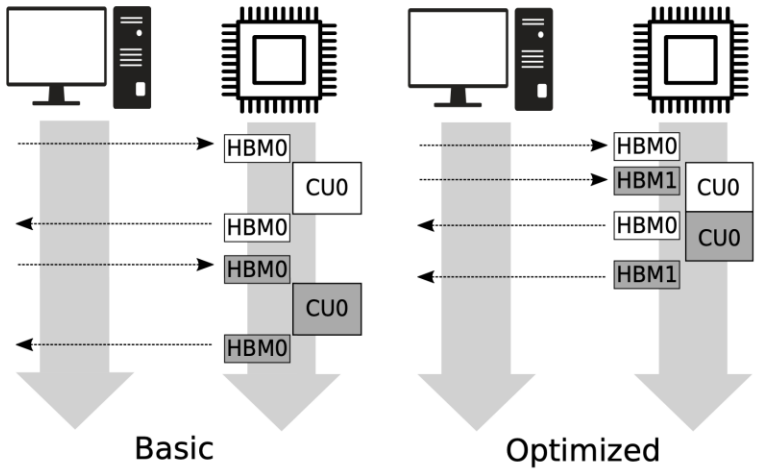
- **Synthesizable C++ code** that includes:
  - **Accelerators** and **PLM** generated with HLS
  - **Communication modules** to match interfaces
    - *Standard AXI interfaces to the system (either cloudFPGA SHELL or HBM channels)*
    - May include **"intelligent" policies** to coordinate (or protect) data transfers
- **System configuration file** to create the overall architecture
  - Support for multiple computing units executing in parallel
  - *Interfacing with Xilinx HLS and synthesis tools*

https://tinyurl.com/CPS2022olympus

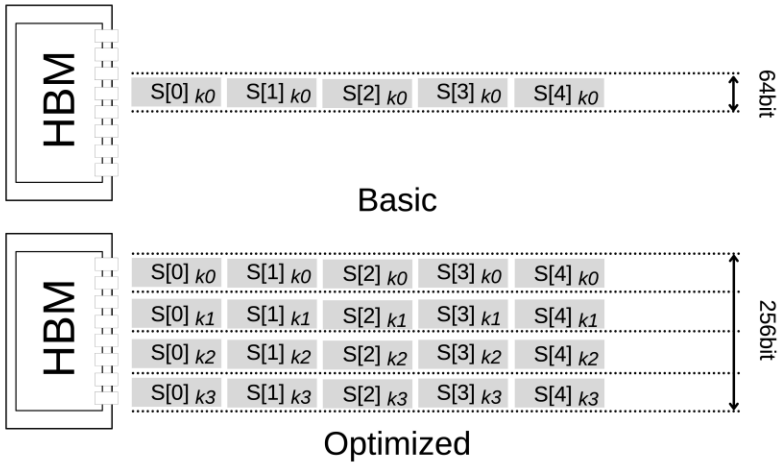
From MLIR to System Architecture

Automatic integration of memory optimizations for **high-performance data transfers**, such as:

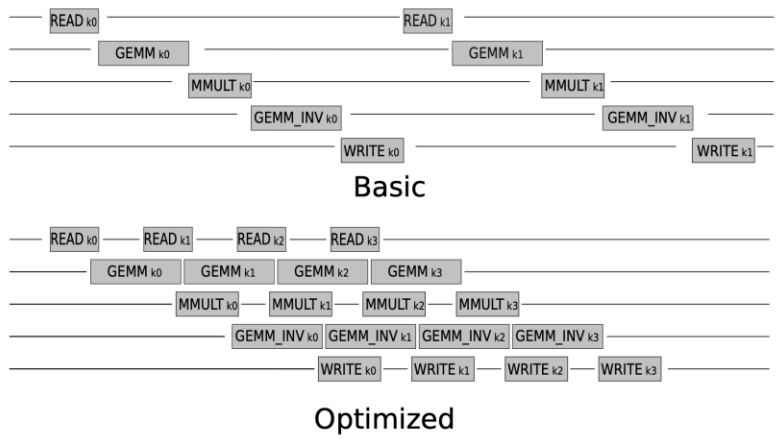
- **Double buffering** to hide latency of host-FPGA data transfers
- **Bus optimization** (and data interleaving) for maximizing bandwidth (e.g., 256-bit AXI channels) – algorithms for efficient data layout on the bus
- **Dataflow execution model** to enable kernel pipelining – automatic (pre-HLS) code transformations



Double buffering



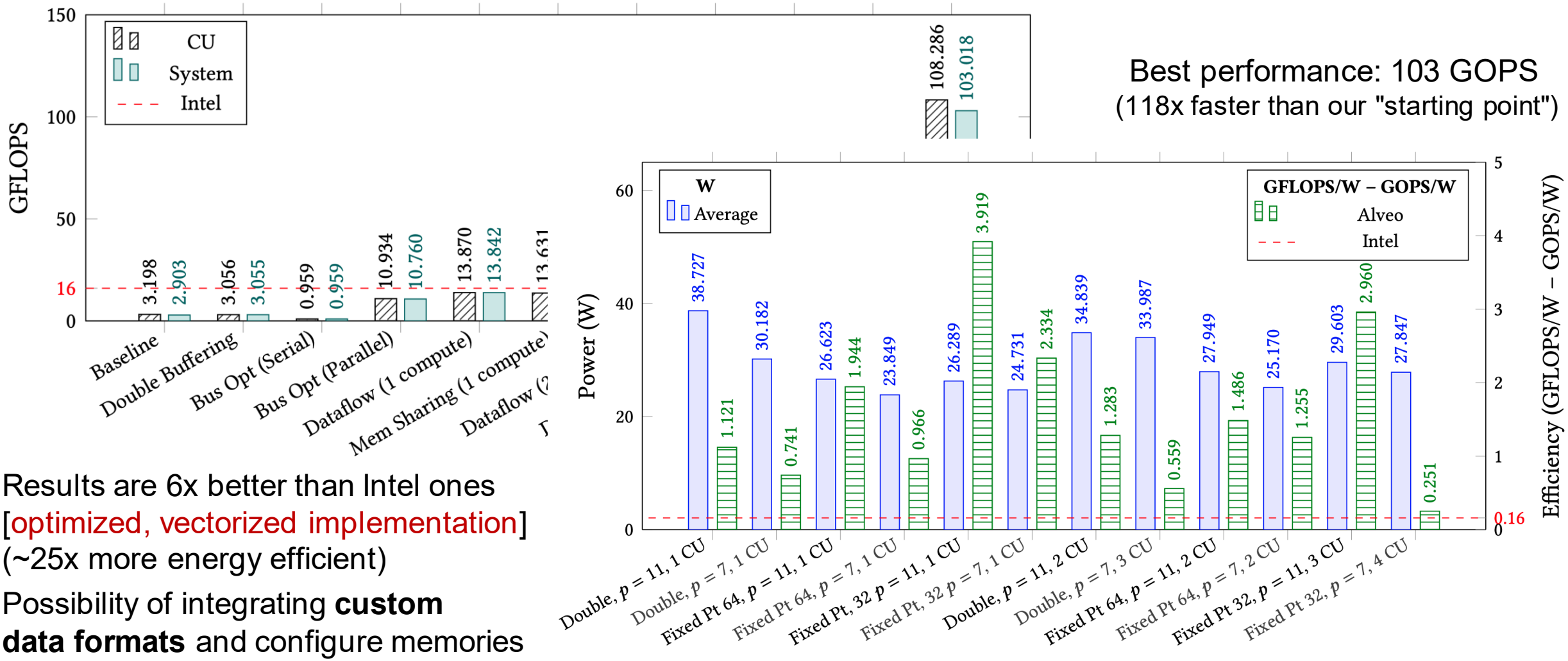
Bus optimization



Dataflow

https://tinyurl.com/CPS2022olympus

Results on HBM FPGA



Results are 6x better than Intel ones [optimized, vectorized implementation] (~25x more energy efficient)

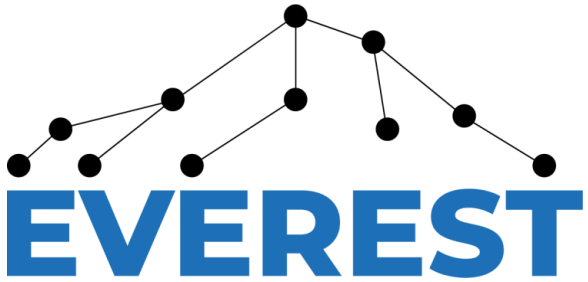
Possibility of integrating **custom data formats** and configure memories and data transfers accordingly

S. Soldavini, K. A. Friebel, M. Tibaldi, G. Hempel, J. Castrillón, C. Pilato:. "Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics" TRETS'22

https://tinyurl.com/CPS2022olympus

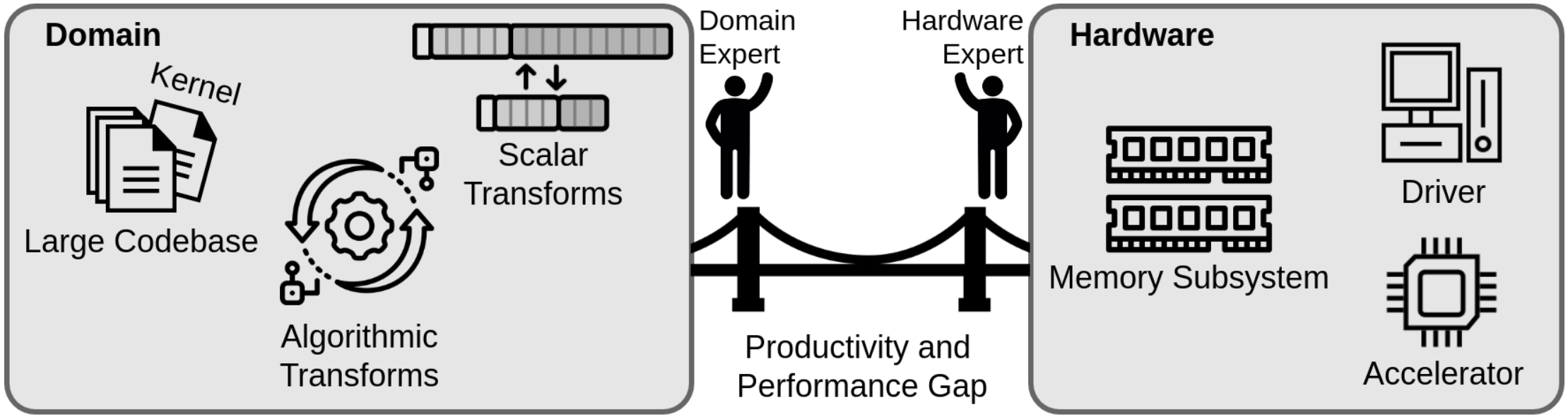
About Me – Stephanie Soldavini

- BS & MS in Computer Engineering from Rochester Institute of Technology (2014-2019)
- PhD Student at Politecnico di Milano (2020-Now)



Problem

- **Productivity**: Application designers usually do not have the necessary hardware design knowledge to create efficient hardware architectures
- **Performance**: Fine-tuned hardware descriptions are required to efficiently coordinate data transfers and execution



Target Boards:
Alveo Data Center Accelerator Cards with HBM

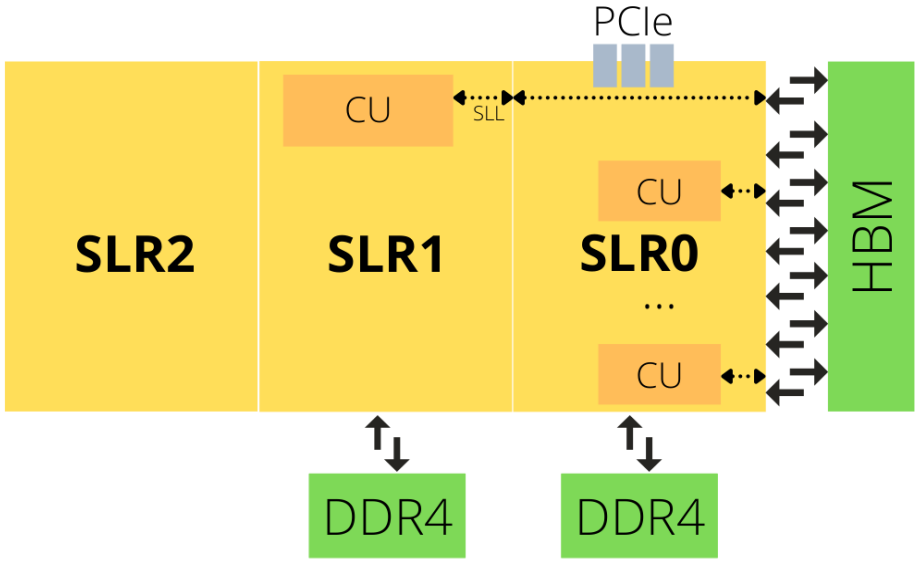
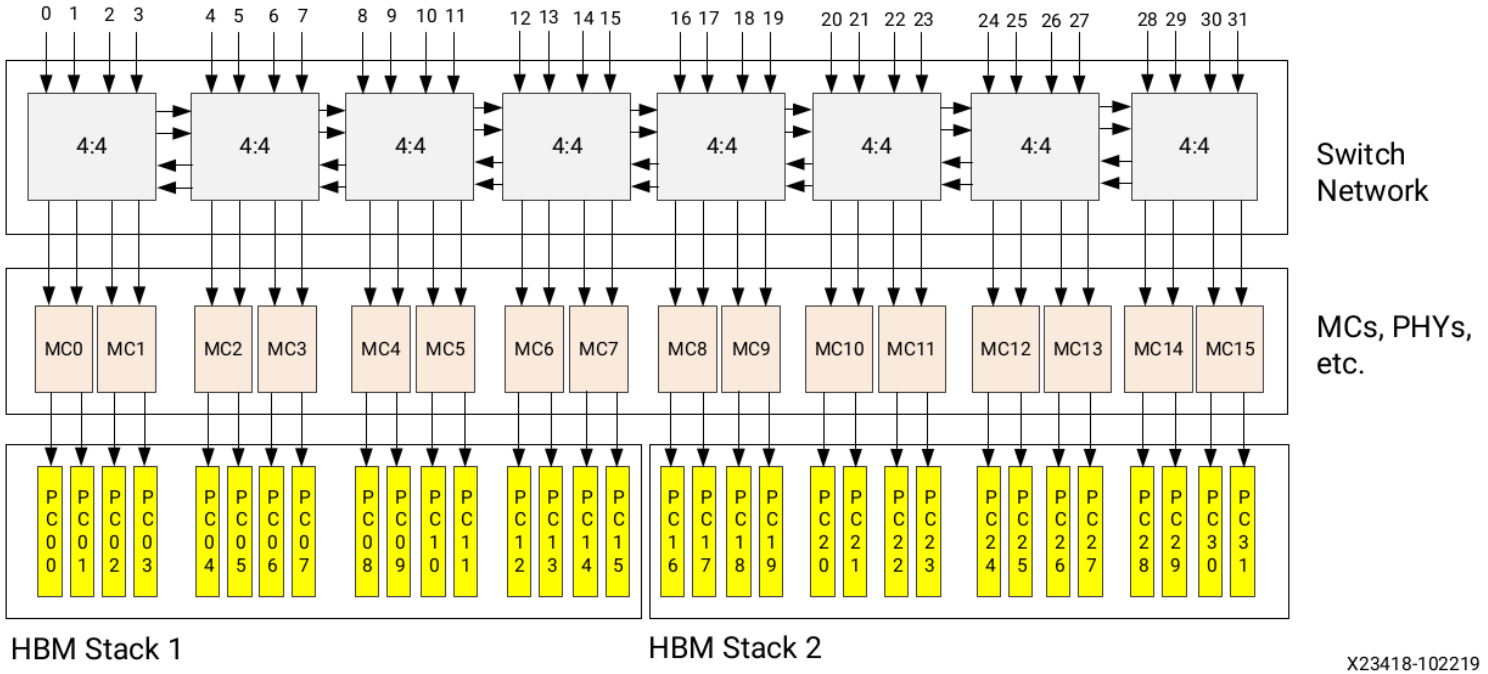


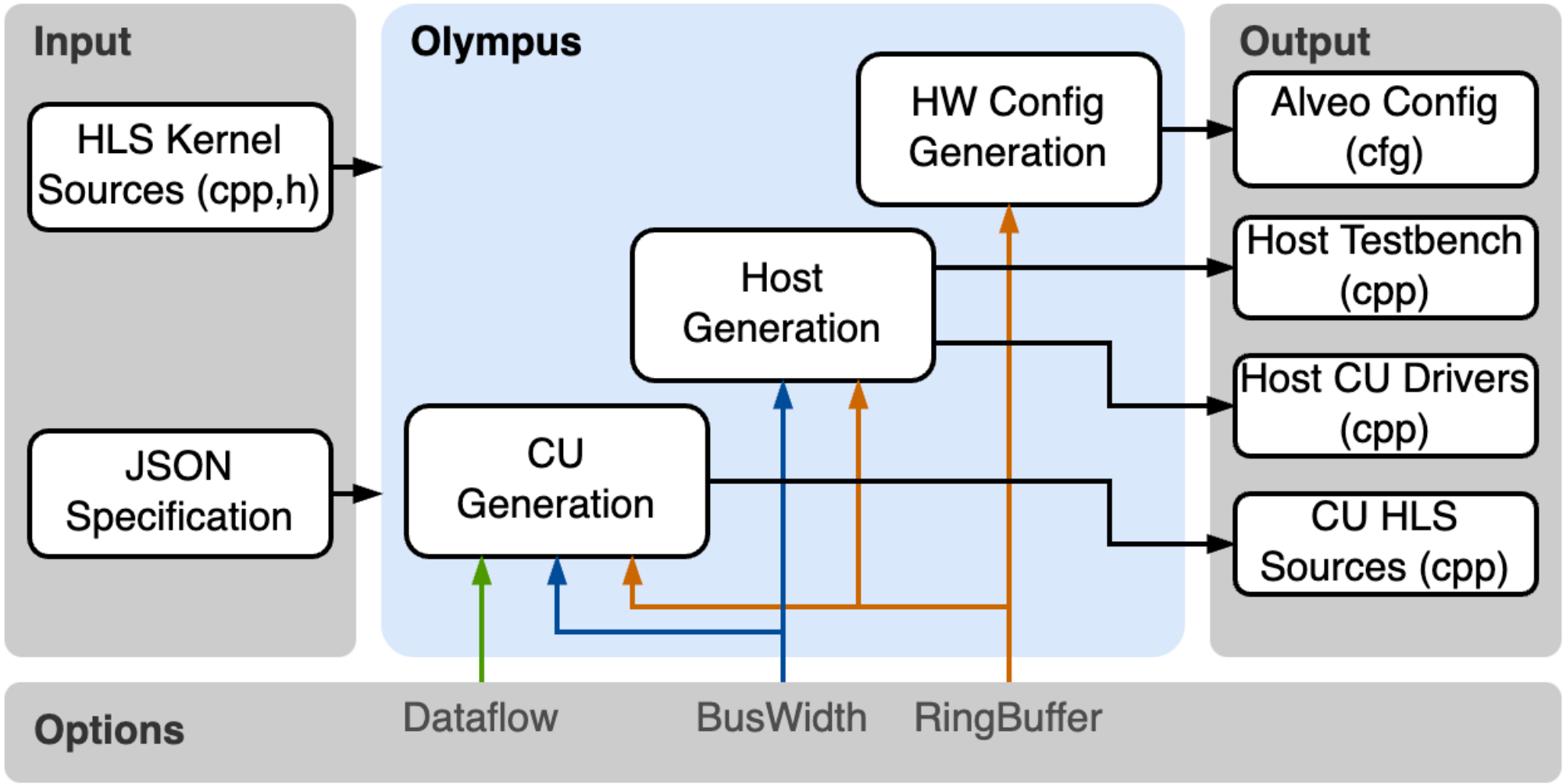
Table 1. Alveo U280 SLR resources.

Resources	SLR0	SLR1	SLR2
HBM	32×256MB	-	-
DDR4	16GB	16GB	-
PLRAM	2×4MB	2×4MB	2×4MB
CLB LUT	369K	333K	367K
CLB Register	746K	675K	729K
Block RAM tile	507	468	512
UltraRAM	320	320	320
DSP	2,733	2,877	2,880



https://tinyurl.com/CPS2022olympus

Olympus Flow Diagram



https://tinyurl.com/CPS2022olympus

Inputs: C HLS Kernel

- C code of application kernel to be accelerated
- Standard C array interfaces

input/kernel_body.cpp

```cpp
#include "kernel_body.h"

void kernel_body(double S[121], double D[1331], double u[1331], double v[1331],
        double t[1331], double r[1331], double t1[1331],
        double t3[1331], double t0[1331], double t2[1331])
{
    for (int c1 = 0; c1 <= 10; c1 += 1)
        for (int c2 = 0; c2 <= 10; c2 += 1)
            for (int c3 = 0; c3 <= 10; c3 += 1) {
                // stmt0
                t1[121 * c1 + 11 * c2 + c3] = 0;
                for (int c8 = 0; c8 <= 10; c8 += 1)
                    // stmt0
                    t1[121 * c1 + 11 * c2 + c3] = t1[121 * c1 + 11 * c2 + c3] + S[11 *
            }
    for (int c1 = 0; c1 <= 10; c1 += 1)
        for (int c2 = 0; c2 <= 10; c2 += 1)
            for (int c3 = 0; c3 <= 10; c3 += 1) {
                // stmt1
                t0[121 * c1 + 11 * c2 + c3] = 0;
                for (int c8 = 0; c8 <= 10; c8 += 1)
                    // stmt1
                    t0[121 * c1 + 11 * c2 + c3] = t0[121 * c1 + 11 * c2 + c3] + S[11 *
            }
    for (int c1 = 0; c1 <= 10; c1 += 1)
        for (int c2 = 0; c2 <= 10; c2 += 1)
            for (int c3 = 0; c3 <= 10; c3 += 1) {
                // stmt2
                t[121 * c1 + 11 * c2 + c3] = 0;
                for (int c8 = 0; c8 <= 10; c8 += 1)
                    // stmt2
                    t[121 * c1 + 11 * c2 + c3] = t[121 * c1 + 11 * c2 + c3] + S[11 * c1
            }
```

Inputs: JSON Kernel Specification

- Format derived from the JSON required for the Vitis RTL Blackbox flow[1]
- Required info:
  - c_function_name
    - the name of the kernel function AND filename (cpp & h)
  - param_type
    - “mm” : memory mapped C arrays
    - “stream” : Xilinx hls::stream<>
  - c_parameters
    - Details on each port interface (must be in the same order as in the C source
    - c_name : Name of the array
    - c_type : Data type of a single element
    - c_port_direction : in, out, or inout
    - depth : Number of elements in the array

input/helmholtz.json

```
{
    "c_function_name": "kernel_body",
    "param_type": "mm",
    "c_parameters": [
        {
            "c_name": "S",
            "c_type": "double",
            "c_port_direction": "in",
            "depth": 121,
        },
        {
            "c_name": "D",
            "c_type": "double",
            "c_port_direction": "in",
            "depth": 1331,
        },
        {
            "c_name": "u",
            "c_type": "double",
            "c_port_direction": "in",
            "depth": 1331,
        },
        {
            "c_name": "v",
            "c_type": "double",
            "c_port_direction": "out",
            "depth": 1331,
        },
        {
            "c_name": "t",
            "c_type": "double",
            "c_port_direction": "inout",
            "depth": 1331
        },
        {
            "c_name": "r",
            "c_type": "double",
            "c_port_direction": "inout",
            "depth": 1331
        },
        {
            "c_name": "t1",
            "c_type": "double",
            "c_port_direction": "inout",
            "depth": 1331
        },
        {
            "c_name": "t3",
            "c_type": "double",
            "c_port_direction": "inout",
            "depth": 1331
        },
        {
            "c_name": "t0",
            "c_type": "double",
            "c_port_direction": "inout",
            "depth": 1331
        },
        {
            "c_name": "t2",
            "c_type": "double",
            "c_port_direction": "inout",
            "depth": 1331
        }
    ]
}
```

[1] https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls/JSON-File-for-RTL-Blackbox

Makefile

- KERNEL_DIR
  - Directory where the kernel HLS sources are
- KERNEL_JSON
  - The JSON spec file path
- RING_BUF
  - The degree of ring buffering
- BUS_WIDTH
  - The bitwidth of the bus to global memory
- STREAMS
  - Whether or not to use a dataflow streaming architecture
  - 1=dataflow, 0=not dataflow
- N_CU
  - Number of CUs to instantiate in the FPGA

```
14 # Olympus options
15 KERNEL_DIR  ?= input/
16 KERNEL_JSON ?= input/helmholtz.json
17 RING_BUF   ?= 2
18 BUS_WIDTH  ?= 64
19 STREAMS    ?= 1
20 N_CU       ?= 1
```

```
15 KERNEL_DIR  ?= input/
16 KERNEL_JSON ?= input/helmholtz.json
17 RING_BUF   ?= 2
18 BUS_WIDTH  ?= 64
19 STREAMS    ?= 1
20 N_CU       ?= 1
21
22 TEST      ?= CLEAN
23
24 #host code
25 HOST_SRC   ?= HostSampleTop.gen.cpp AlveoHost.cpp HostImpl.gen.cpp ../src/$(KERNEL_MODEL).cpp
26 EXEC_FLAGS ?=
27 #kernel code
28 CU_SRC      ?= CU.cpp $(KERNEL_BODY).cpp
29 CU_NAME     ?= krnl_helm
30 KFLAGS      ?=
31 #run flags
32 RUN_FLAGS   ?= $(N_CU) $(POINTS)
33
34 #design configuration
35 CFG         ?= Design.cfg
36
37 include $(BOARD_TARGET_PATH)/targets.Makefile
```


EVEREST

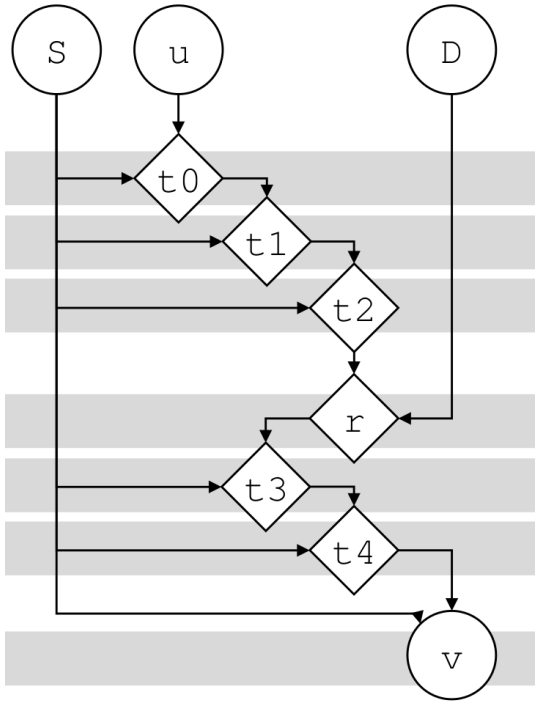
Test Kernel: Inverse Helmholtz

- Tensor operator commonly used in computational fluid dynamics (CFD) applications
- Implemented in C as 7 loop nests of depth 3-4
  - `input/kernel_body.cpp`

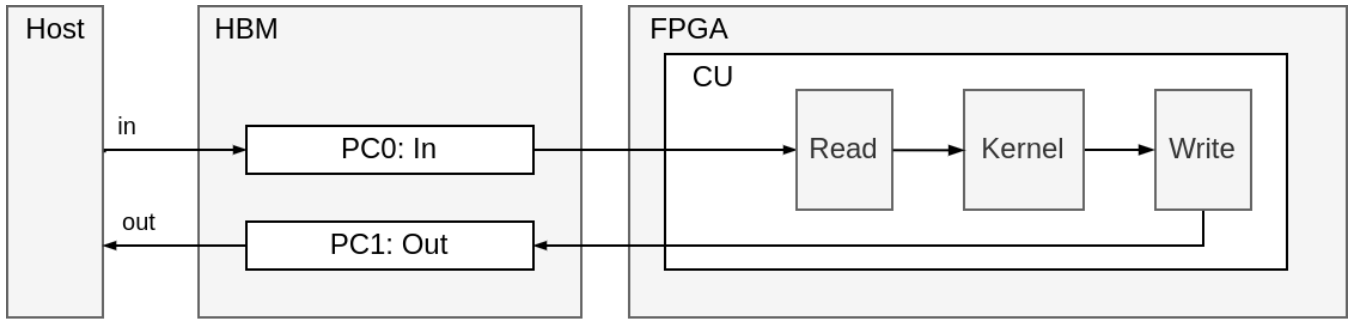
$$t_{ijk,e} = \sum_{l=0}^{p}\sum_{m=0}^{p}\sum_{n=0}^{p} S^T_{li} \cdot S^T_{mj} \cdot S^T_{nk} \cdot u_{lmn,e} = (S \otimes S \otimes S \otimes u_e)_{iljmkn}{}^{lmn} \quad (1a)$$

$$r_{ijk,e} = D_{ijk,e} \cdot t_{ijk,e} \quad (1b)$$

$$v_{ijk,e} = \sum_{l=0}^{p}\sum_{m=0}^{p}\sum_{n=0}^{p} S_{li} \cdot S_{mj} \cdot S_{nk} \cdot r_{lmn,e} = (S \otimes S \otimes S \otimes r_e)_{limjnk}{}^{lmn} \quad (1c)$$



Basic Implementation



- `make olympus`
  - Generate the code (default values in Makefile yield basic implementation)
  - Sources: `~/alveo_tests/helmholtz_autogen/RB1_BW64_S0-student/krnl_helm/CLEAN/`
    - `src/` : kernel sources, `CU.cpp` is the generated Compute Unit wrapper
    - `host/` : host sources
      - `HostImpl.gen.cpp`: Implementation of driver functions (moving data to global mem, invoking CU)
      - `HostSampleTop.gen.cpp`: A sample test bench `main` file
- `make chost chw POINTS=16 run`
  - Build the sw_emu versions of the host and hardware and run with 16 iterations
- `make hls TARGET=hw`
  - Run HLS

Basic - Results

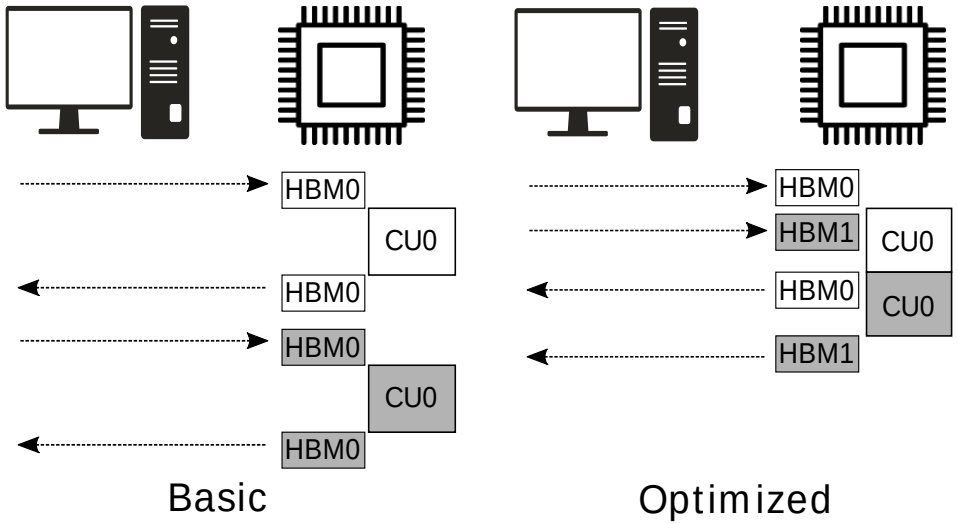
`source /tools/Xilinx/Vitis/2021.1/settings64.sh`

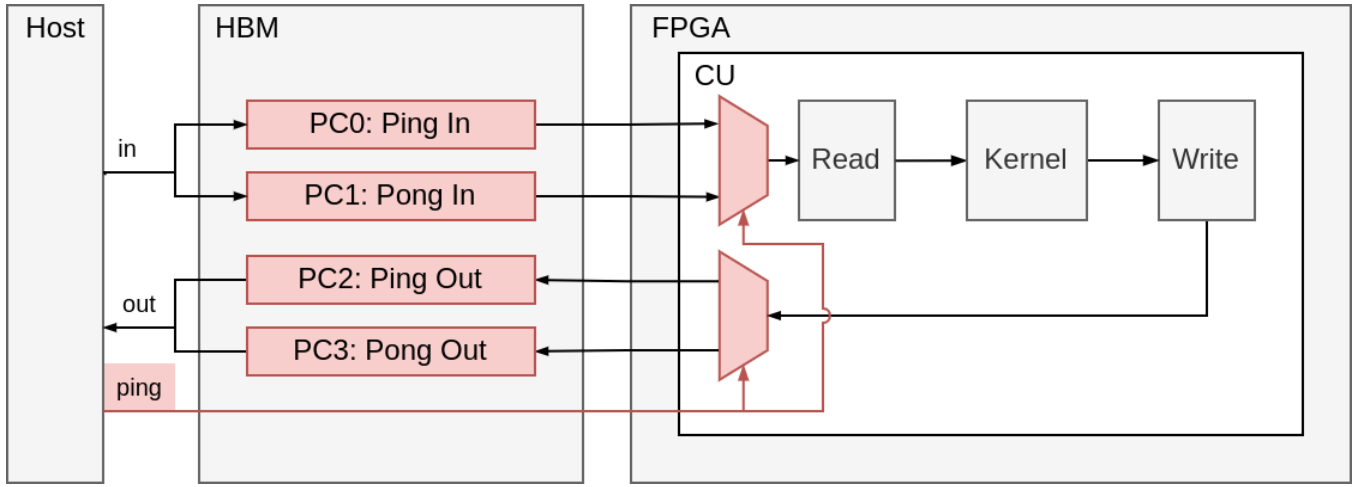
3rd-to-last line of "`make hls`" output: `vitis_analyzer [path]/[kernel name].xo.compile_summary`

Name	Latency (cycles)	Iteration Latency	Interval
krnl_helm			
krnl_helm_Pipeline_1	124		124
krnl_helm_Pipeline_2	1334		1334
krnl_helm_Pipeline_3	1334		1334
compute_1	9744		9744
compute_1_Pipeline_VITIS_LOOP_7_1_VITIS_LOOP_8_2_VITIS_LOOP_9_3	1398		1398
compute_1_Pipeline_VITIS_LOOP_16_5_VITIS_LOOP_17_6_VITIS_LOOP_18_7	1398		1398
compute_1_Pipeline_VITIS_LOOP_25_9_VITIS_LOOP_26_10_VITIS_LOOP_27_11	1398		1398
compute_1_Pipeline_VITIS_LOOP_34_13_VITIS_LOOP_35_14_VITIS_LOOP_36_15	1343		1343
compute_1_Pipeline_VITIS_LOOP_39_16_VITIS_LOOP_40_17_VITIS_LOOP_41_18	1398		1398
compute_1_Pipeline_VITIS_LOOP_48_20_VITIS_LOOP_49_21_VITIS_LOOP_50_22	1398		1398
compute_1_Pipeline_VITIS_LOOP_57_24_VITIS_LOOP_58_25_VITIS_LOOP_59_26	1398		1398
krnl_helm_Pipeline_4	1334		1334
VITIS_LOOP_41_1		14159	

BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
10	~0	135	1	40078	1	27879	2

Ping Pong Buffers





- `make olympus RING_BUF=2`
  - RING_BUF sets the degree of ring buffering, 2 => ping pong
  - Sources: `~/alveo_tests/helmholtz_autogen/RB2_BW64_S0-student/krnl_helm/CLEAN/`
- `make chost chw RING_BUF=2 POINTS=16 run`
  - Build the sw_emu versions of the host and hardware and run with 16 iterations
- `make hls RING_BUF=2 TARGET=hw`
  - Run HLS

Ping Pong Buffers - Results

Name	Latency (cycles)	Iteration Latency	Interval
krnl_helm			
krnl_helm_Pipeline_4	124		124
krnl_helm_Pipeline_5	1334		1334
krnl_helm_Pipeline_6	1334		1334
krnl_helm_Pipeline_1	124		124
krnl_helm_Pipeline_2	1334		1334
krnl_helm_Pipeline_3	1334		1334
compute_1	9744		9744
compute_1_Pipeline_VITIS_LOOP_7_1_VITIS_LOOP_8_2_VITIS_LOOP_9_3	1398		1398
compute_1_Pipeline_VITIS_LOOP_16_5_VITIS_LOOP_17_6_VITIS_LOOP_18_7	1398		1398
compute_1_Pipeline_VITIS_LOOP_25_9_VITIS_LOOP_26_10_VITIS_LOOP_27_11	1398		1398
compute_1_Pipeline_VITIS_LOOP_34_13_VITIS_LOOP_35_14_VITIS_LOOP_36_15	1343		1343
compute_1_Pipeline_VITIS_LOOP_39_16_VITIS_LOOP_40_17_VITIS_LOOP_41_18	1398		1398
compute_1_Pipeline_VITIS_LOOP_48_20_VITIS_LOOP_49_21_VITIS_LOOP_50_22	1398		1398
compute_1_Pipeline_VITIS_LOOP_57_24_VITIS_LOOP_58_25_VITIS_LOOP_59_26	1398		1398
krnl_helm_Pipeline_8	1334		1334
krnl_helm_Pipeline_7	1334		1334
VITIS_LOOP_62_1		14159	

Previous:

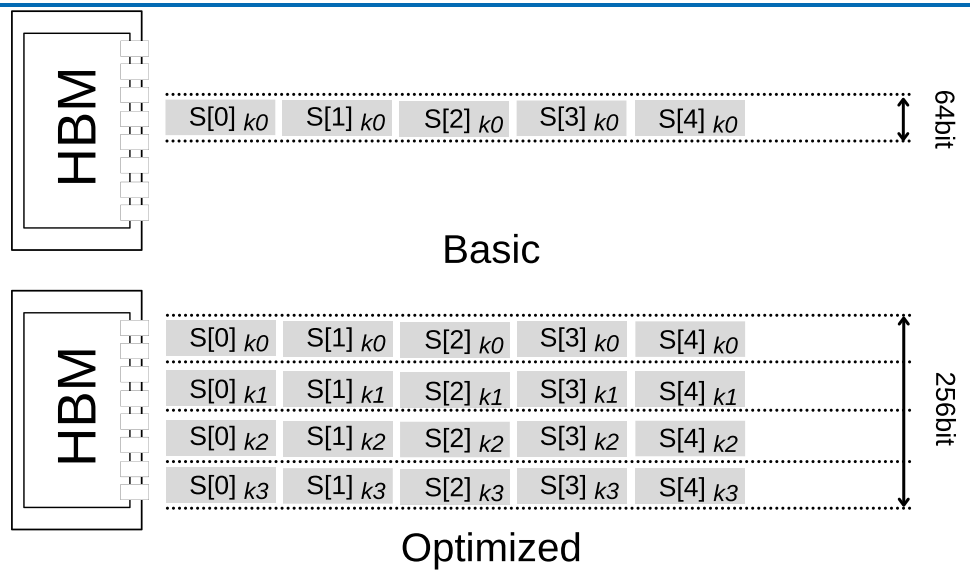
BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
10	~0	135	1	40078	1	27879	2

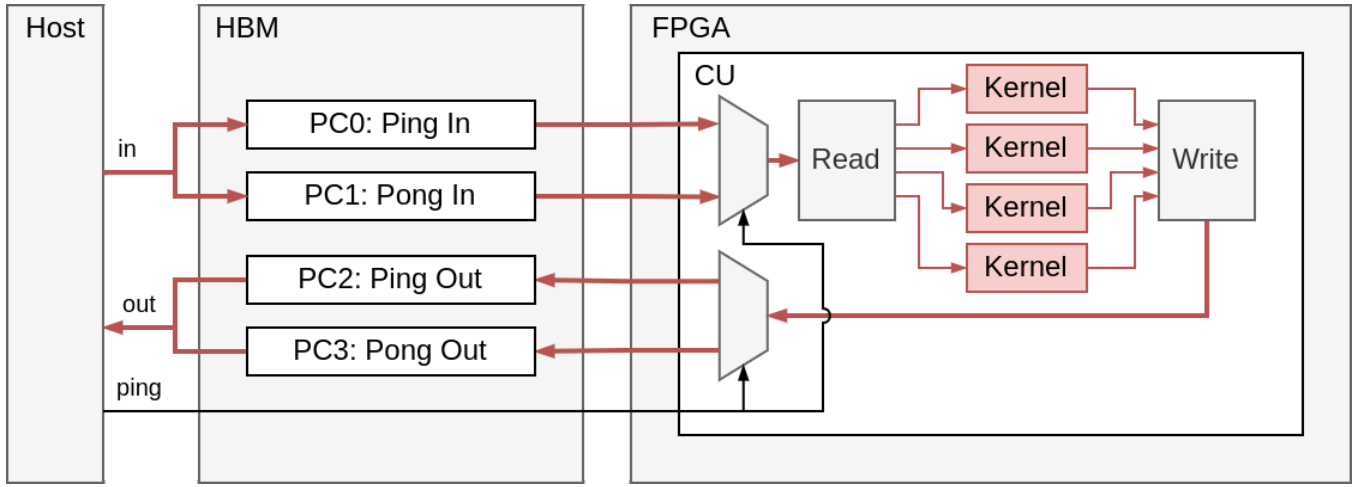
Now:

BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
18	~0	135	1	42747	1	32108	2

Interface	Data Width (SW->HW)
m_axi_gmem0	64 -> 64
m_axi_gmem1	64 -> 64
m_axi_gmem2	64 -> 64
m_axi_gmem3	64 -> 64

Wide Bus





- `make olympus RING_BUF=2 BUS_WIDTH=256`
  - BUS_WIDTH sets the width of the bus to global memory. 256/sizeof(double) => 4 "lanes"
  - Sources: `~/alveo_tests/helmholtz_autogen/RB2_BW256_S0-student/krnl_helm/CLEAN/`
- `make chost chw RING_BUF=2 BUS_WIDTH=256 POINTS=16 run`
  - Build the sw_emu versions of the host and hardware and run with 16 iterations
- `make hls RING_BUF=2 BUS_WIDTH=256 TARGET=hw`
  - Run HLS

Wide Bus - Results

Name	Issue Type	Latency (cycles)	Iteration Latency	Interval
krnl_helm				
read_data_1		3009		3009
compute		49659		49659
compute_Pipeline_VITIS_LOOP_7_1_VITIS_LOOP_8_2_VITIS_LOOP_9_3	II Violation	8051		8051
compute_Pipeline_VITIS_LOOP_16_5_VITIS_LOOP_17_6_VITIS_LOOP_18_7	II Violation	8051		8051
compute_Pipeline_VITIS_LOOP_25_9_VITIS_LOOP_26_10_VITIS_LOOP_27_11	II Violation	8051		8051
compute_Pipeline_VITIS_LOOP_34_13_VITIS_LOOP_35_14_VITIS_LOOP_36_15		1343		1343
compute_Pipeline_VITIS_LOOP_39_16_VITIS_LOOP_40_17_VITIS_LOOP_41_18	II Violation	8050		8050
compute_Pipeline_VITIS_LOOP_48_20_VITIS_LOOP_49_21_VITIS_LOOP_50_22	II Violation	8050		8050
compute_Pipeline_VITIS_LOOP_57_24_VITIS_LOOP_58_25_VITIS_LOOP_59_26	II Violation	8050		8050
krnl_helm_Pipeline_VITIS_LOOP_133_2		1334		1334
krnl_helm_Pipeline_VITIS_LOOP_117_1		1334		1334
VITIS_LOOP_188_1			54077	

Previous:

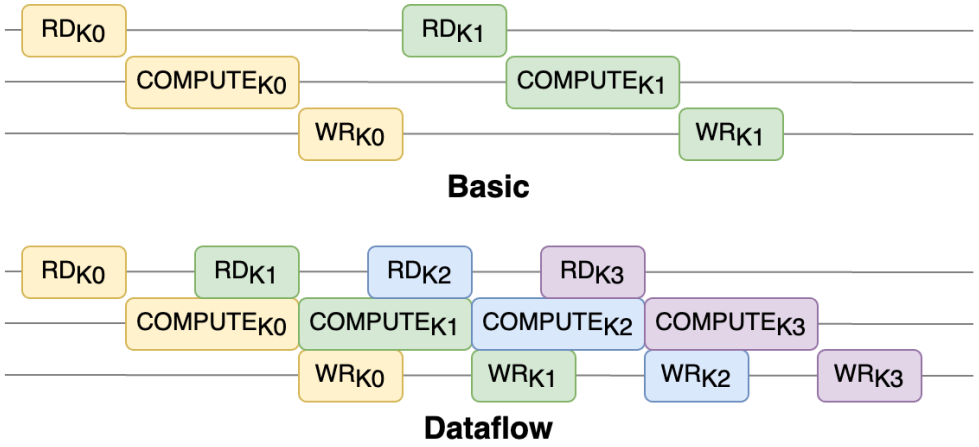
BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
18	~0	135	1	42747	1	32108	2

Now:

BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
72	1	142	1	109117	4	66257	5

Interface	Data Width (SW->HW)
m_axi_gmem0	256 -> 256
m_axi_gmem1	256 -> 256
m_axi_gmem2	256 -> 256
m_axi_gmem3	256 -> 256

Streaming (1 compute)



- `make olympus RING_BUF=2 BUS_WIDTH=256 STREAMS=1`
  - STREAMS=1 turns on the HLS dataflow pragma and uses the hls::stream data type
  - Sources: `~/alveo_tests/helmholtz_autogen/RB2_BW256_S1-student/krnl_helm/CLEAN/`
- `make chost chw RING_BUF=2 BUS_WIDTH=256 STREAMS=1 POINTS=16 run`
  - Build the sw_emu versions of the host and hardware and run with 16 iterations
- `make hls RING_BUF=2 BUS_WIDTH=256 STREAMS=1 TARGET=hw`
  - Run HLS

Streaming (1 compute) - Results

Name	Latency (cycles)	Iteration Latency	Interval
krnl_helm			
dataflow_in_loop_VITIS_LOOP_229_1	14088		13877
read_data_1	3009		3009
compute_11	13876		13876
compute_12	13876		13876
compute_13	13876		13876
compute_1	13876		13876
compute_1_Pipeline_VITIS_LOOP_162_1	123		123
compute_1_Pipeline_VITIS_LOOP_167_2	1333		1333
compute_1_Pipeline_VITIS_LOOP_172_3	1333		1333
compute_1_Pipeline_VITIS_LOOP_7_1_VITIS_LOOP_8_2_VITIS_LOOP_9_3	1398		1398
compute_1_Pipeline_VITIS_LOOP_16_5_VITIS_LOOP_17_6_VITIS_LOOP_18_7	1398		1398
compute_1_Pipeline_VITIS_LOOP_25_9_VITIS_LOOP_26_10_VITIS_LOOP_27_11	1398		1398
compute_1_Pipeline_VITIS_LOOP_34_13_VITIS_LOOP_35_14_VITIS_LOOP_36_15	1343		1343
compute_1_Pipeline_VITIS_LOOP_39_16_VITIS_LOOP_40_17_VITIS_LOOP_41_18	1398		1398
compute_1_Pipeline_VITIS_LOOP_48_20_VITIS_LOOP_49_21_VITIS_LOOP_50_22	1398		1398
compute_1_Pipeline_VITIS_LOOP_57_24_VITIS_LOOP_58_25_VITIS_LOOP_59_26	1398		1398
compute_1_Pipeline_VITIS_LOOP_178_4	1333		1333
entry_proc	0		0
write_data_1	1407		1407
VITIS_LOOP_229_1			

Previous "Compute" Latency: 49659

Previous Iteration Latency: 54077

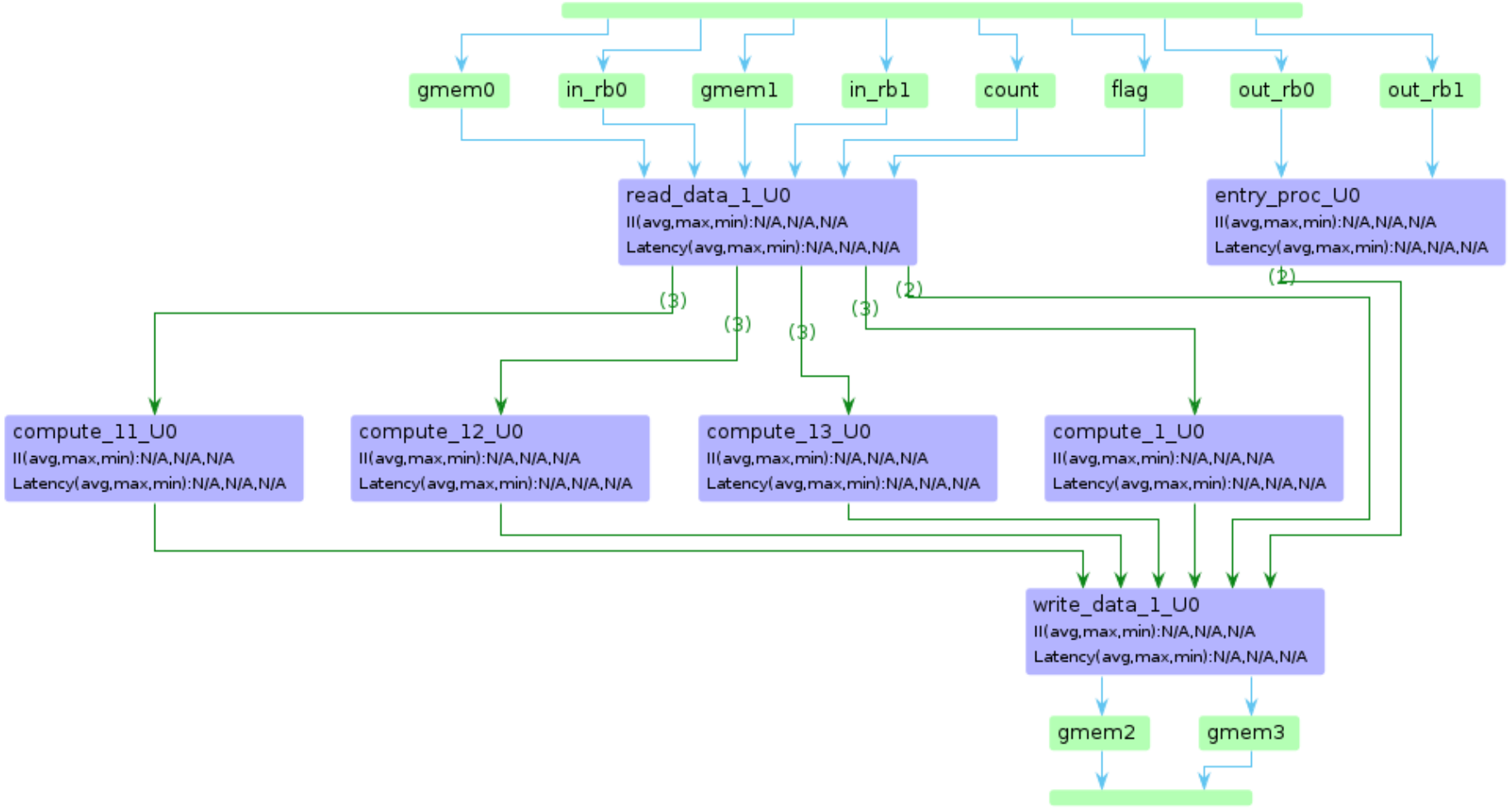
Previous:

BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
72	1	142	1	109117	4	66257	5

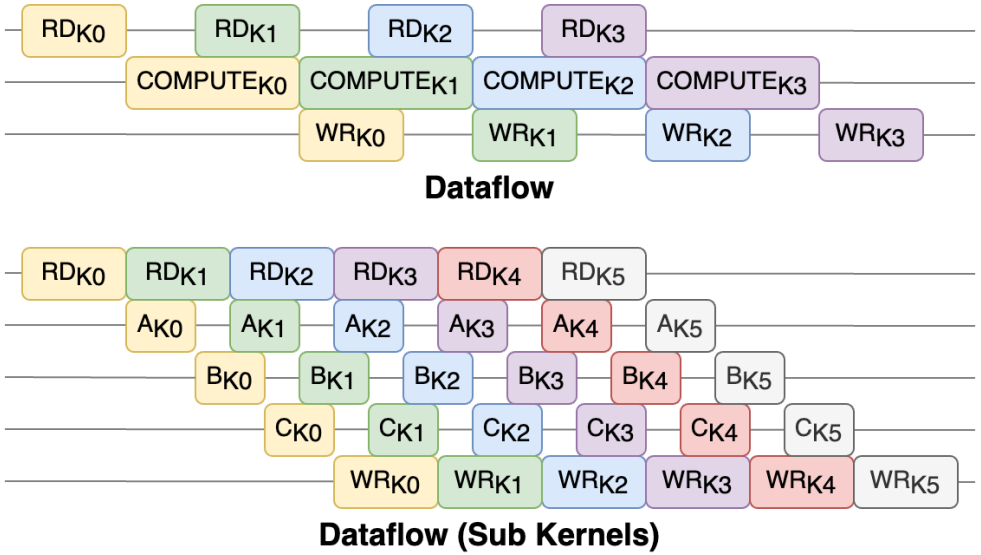
Now:

BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
104	2	540	5	158002	6	105005	8

Streaming (1 compute) - Results



Streaming (multi compute)



- Multi compute is not controlled by Olympus, but using the INLINE pragma any submodules in the original HLS kernel will be used in the pipeline
- Edit the Makefile:
  - `KERNEL_BODY ?= kernel_body_df`
  - `KERNEL_MODEL ?= kernel_body_sw`
  - `KERNEL_JSON ?= input/helmholtz_df.json`
  - `TEST ?= DF`

- Same commands as before:
  - `make olympus RING_BUF=2 BUS_WIDTH=256 STREAMS=1`
    - Sources: `~/alveo_tests/helmholtz_autogen/RB2_BW256_S1-student/krnl_helm/DF/`
  - `make chost chw RING_BUF=2 BUS_WIDTH=256 STREAMS=1 POINTS=16 run`
  - `make hls RING_BUF=2 BUS_WIDTH=256 STREAMS=1 TARGET=hw`

Streaming (multi compute) - Results

Name	Latency (cycles)	Iteration Latency	Interval
krnl_helm			
dataflow_in_loop_VITIS_LOOP_199_1	3429		3010
read_data_1	3009		3009
gemm5	2861		2861
...			
gemm	2861		2861
mmult8	1340		1340
mmult15	1340		1340
mmult22	1340		1340
mmult	1340		1340
gemm_inv9	2861		2861
...			
gemm_inv_last	2861		2861
entry_proc	0		0
write_data_1	1407		1407
VITIS_LOOP_199_1			

Previous Dataflow Interval: 13877

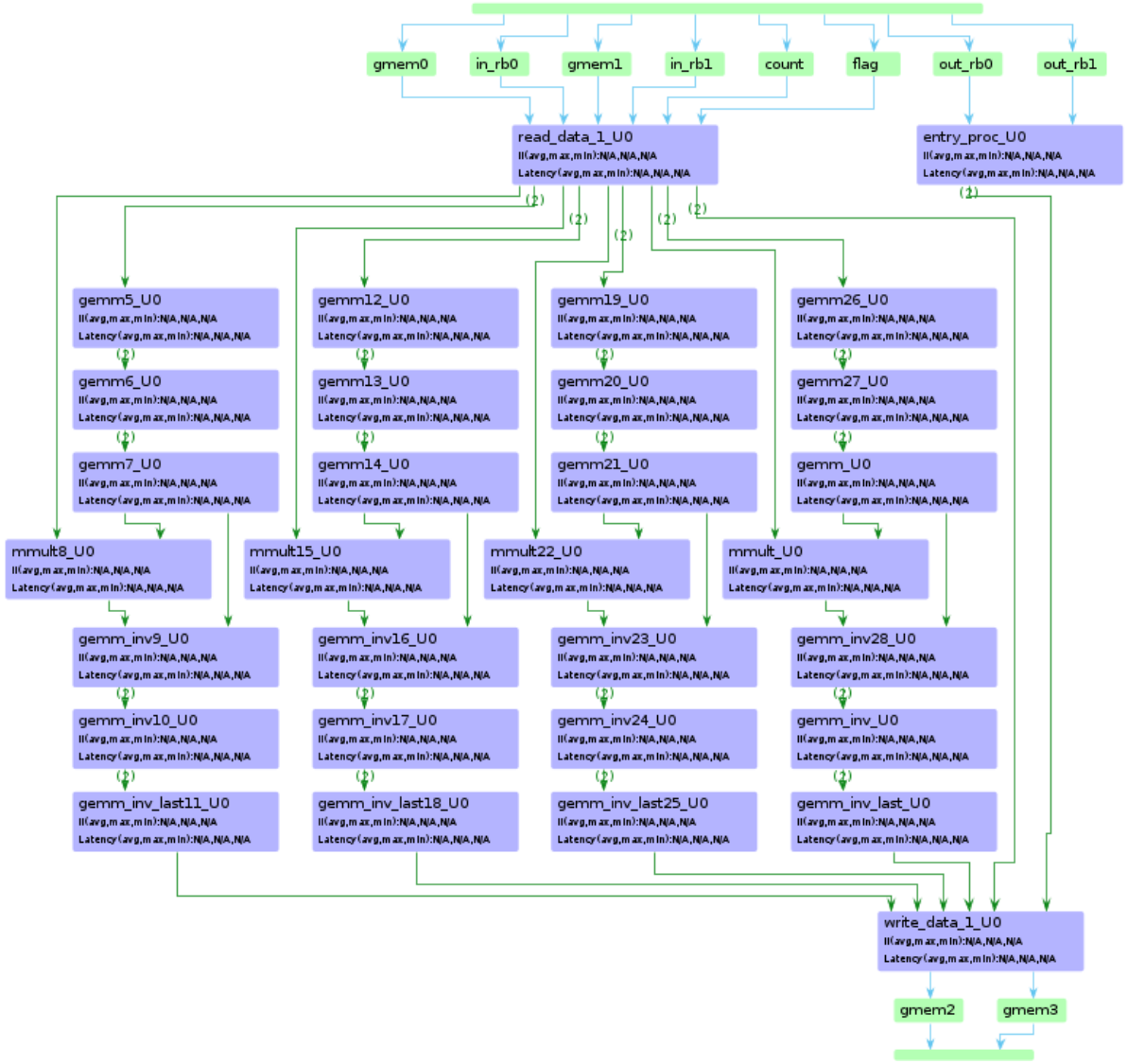
Previous:

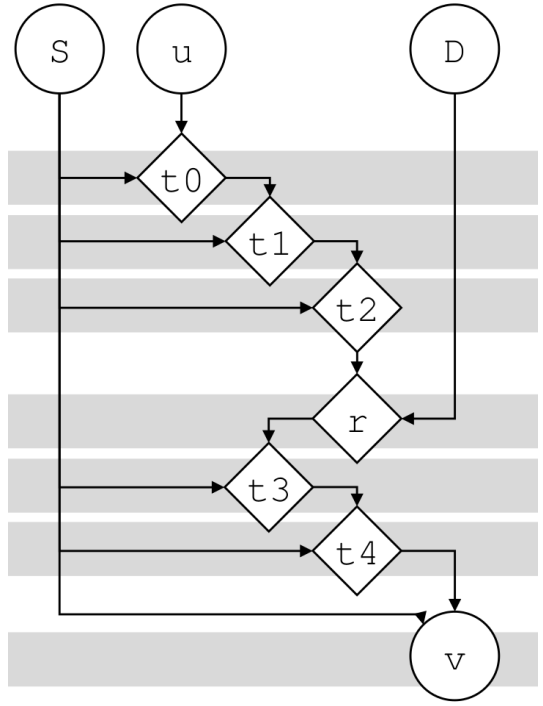
BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
104	2	540	5	158002	6	105005	8

Now:

BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
144	3	2964	32	344814	13	279029	21

Streaming (multi compute) - Results





Conclusions

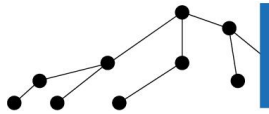
Data management optimizations are becoming the key for the creation of **efficient FPGA architectures** (… more than pure kernel optimizations)

HLS is now used not only to create accelerator kernels but also to generate the **system-level architecture**

- **Portable solutions** across multiple target platforms

Novel **HBM architectures** offer high bandwidth (that's why they are called *high-bandwidth memory* architectures… ☺) but their design is complex:

- Necessary to match **application requirements** and **technology characteristics**
- We propose an **MLIR-based compilation flow** that directly interfaces with **commercial HLS tools**



EVEREST

DESIGN ENVIRONMENT
FOR EXTREME-SCALE BIG DATA ANALYTICS
ON HETEROGENEOUS PLATFORMS

Thank you!

CHRISTIAN PILATO

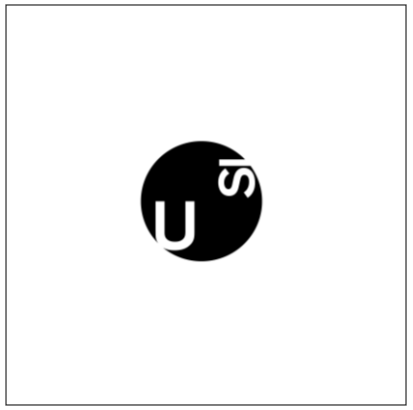
EVEREST Scientific Coordinator
Assistant Professor, Politecnico di Milano

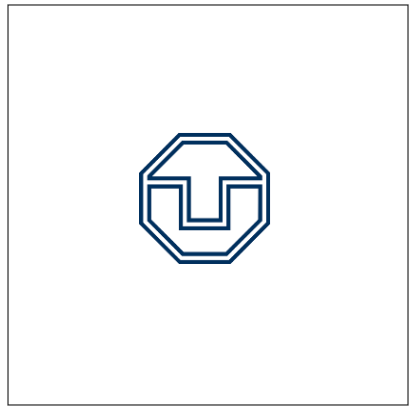
christian.pilato@polimi.it

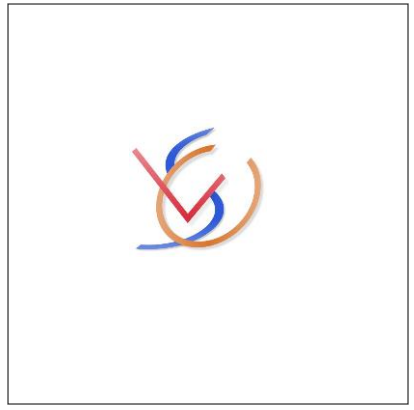
STEPHANIE SOLDAVINI

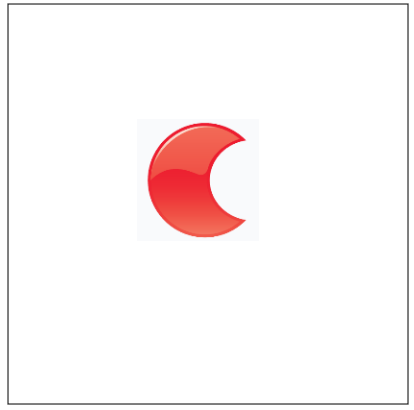
PhD Student, Politecnico di Milano

stephanie.soldavini@polimi.it









This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957269