POLITECNICO
MILANO 1863

EVEREST

New York University – May 11, 2022

Generating HPC memory architectures with HLS: The two sides of the medal

Christian Pilato

Assistant Professor

christian.pilato@polimi.it

About Me

Assistant Professor (RTD-B - Ricercatore a Tempo Determinato Senior)

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

PhD Student
2008-2011

Research Assistant
2011-2013

Postdoc Research Scientist
2013-2016

Postdoc Research Assistant
2016-2018

Assistant Prof.
2018-*now*

R&D Internship
6 months

Visiting Researcher
3 months

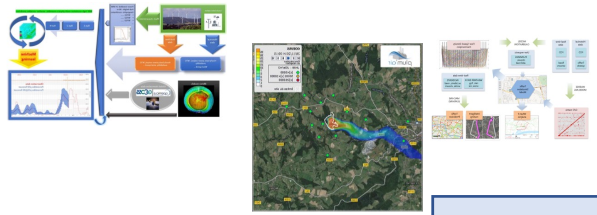
Visiting Researcher
4 months

Visiting Researcher
9 months

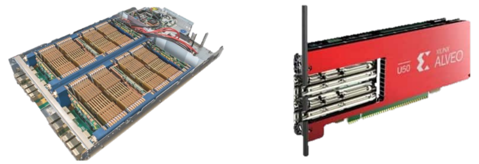
R&D Projects

- FP6 HARTES
- FP7 SYNAPTIC
- FP7 FASTER
- DARPA PERFECT
- SRC CFAR
- H2020 CERBERO
- DARPA CRAFT
- H2020 EVEREST

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

EVEREST

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

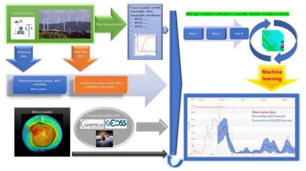
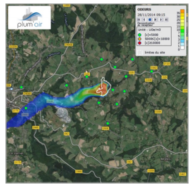
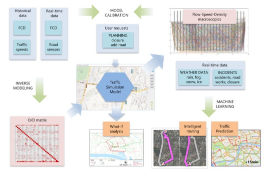
Multi-node support

POLITECNICO MILANO 1863

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

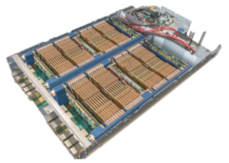
Open-source framework to support the optimization of selected workflow tasks

FPGA-based architectures to accelerate selected kernels

CPU-based infrastructure

+

Two FPGA-based clusters



EVEREST Partners

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

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

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

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

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

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

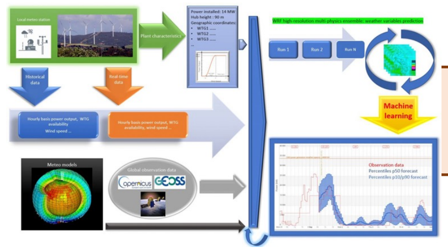
Virtual Open Systems (France)
Virtualization techniques, runtime extensions to manage heterogeneous resources
PI: Michele Paolino

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

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

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

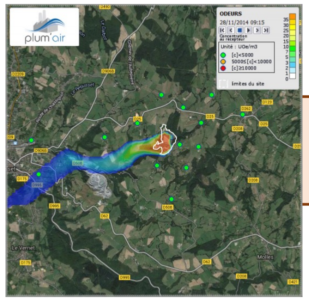
EVEREST Use Cases



Renewable energy production prediction

★ Improve **quality of the predictions**

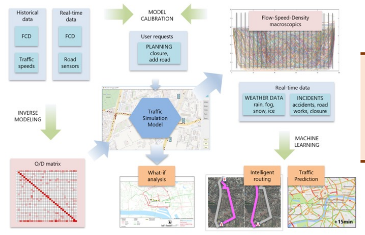
Weather prediction modelling (WRF)



Air-quality monitoring of industrial sites

★ Improve the **response time of predictions**

★ Accelerate kernels to execute more tests



Traffic modeling for intelligent transportation

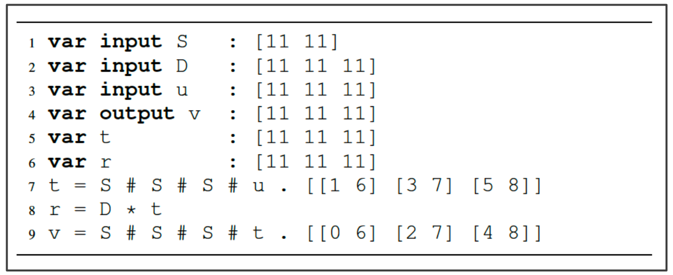
★ Improve the **overall performance of traffic simulation**

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

The Case of Computational Fluid Dynamics

Numerical simulations are becoming more and more popular for many applications

- **Computational Fluid Dynamics** (CFD) is a representative application that requires to solve partial differential equations
- Kernel is the **Inverse Helmholtz operator** (parametric with respect to polynomial degree *p*) – “Helmholtz” for the friends



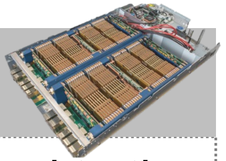
```
var input S    : [11 11]
var input D    : [11 11 11]
var input u    : [11 11 11]
var output v   : [11 11 11]
var t          : [11 11 11]
var r          : [11 11 11]
t = S # S # S # u . [[1 6] [3 7] [5 8]]
r = D ⋆ t
v = S # S # S # t . [[0 6] [2 7] [4 8]]
```

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
- It requires additional *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**

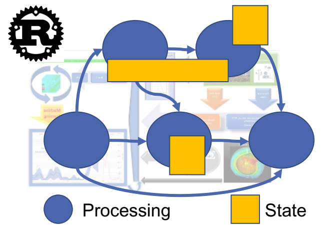
Different nodes to better **match applications**

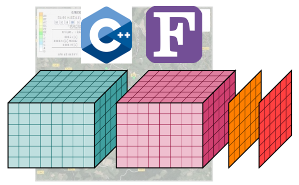
Data-intensive (memory-bound) applications

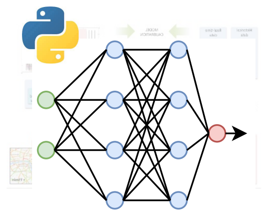
Seamless support for multiple nodes

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

EVEREST System Development Kit (SDK)

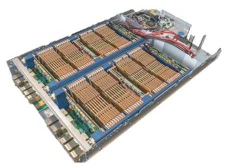






Different **input flows**
starting from different **input languages**

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

PandA

XILINX VITIS

cFDK

HEAppE

HyperLoom

(and more...)

Challenges for HPC Architectures (i)

We can identify common challenges to most of the FPGA-based HPC architectures (e.g., network-attached cloudFPGA or bus-attached Alveo)

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 – 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

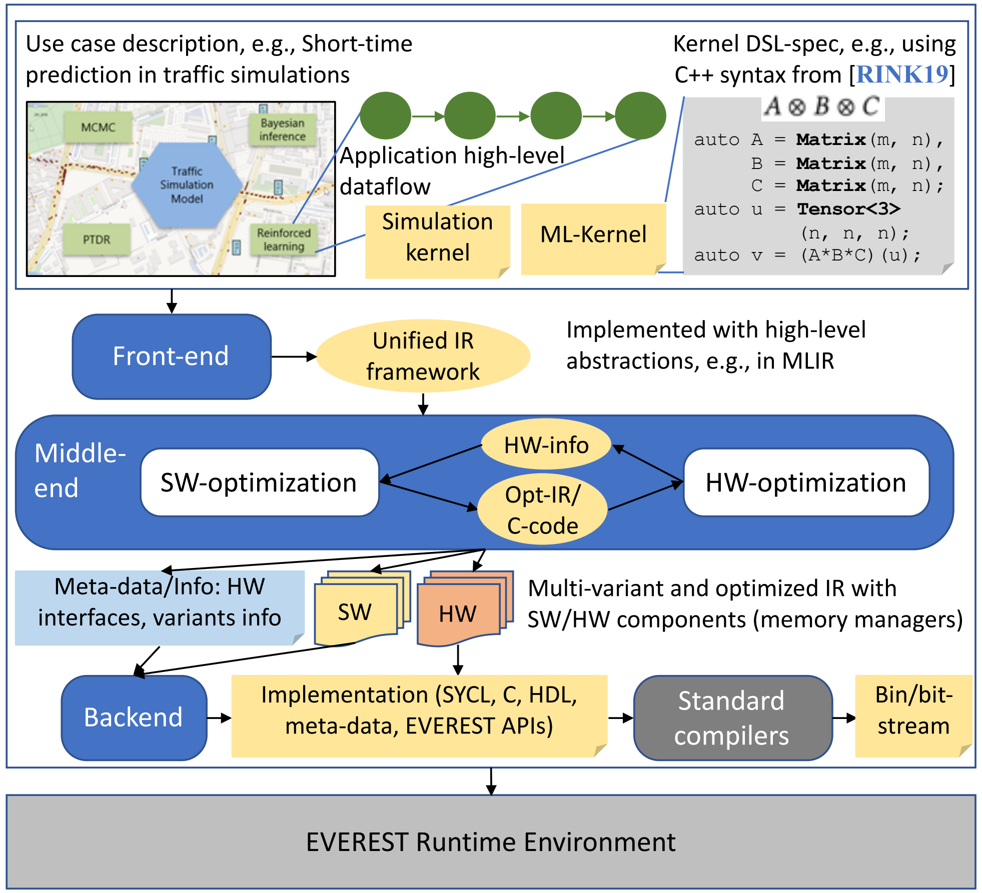
- Data must be placed in memory to maximize its utilization but also to enable efficient data transfers/computation – custom data layouts?

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 (or architectures) may be different – how to separate platform-agnostic and platform-dependent parts?
- FPGA logic architectures are complex and may introduce performance degradation – how to "guide" the synthesis process?

EVEREST Programming Environment

1. **Compilation Environment:** analyzes application and creates all "variants" based on <u>*architecture abstraction*</u> and <u>*application/data requirements*</u>
   - **Exploring unified IR framework** (e.g., **MLIR**)
   - **Integration of non-functional properties with domain-specific extensions**
   - **Hardware acceleration and High-level synthesis** (**Bambu**, **Vivado HLS**)


Use case description, e.g., Short-time prediction in traffic simulations
MCMC
Bayesian inference
Traffic Simulation Model
PTDR
Reinforced learning
Application high-level dataflow
Simulation kernel
ML-Kernel
Kernel DSL-spec, e.g., using C++ syntax from [RINK19]
$A \otimes B \otimes C$
Front-end
Unified IR framework
Implemented with high-level abstractions, e.g., in MLIR
Middle-end
SW-optimization
HW-info
Opt-IR/ C-code
HW-optimization
Meta-data/Info: HW interfaces, variants info
SW
HW
Multi-variant and optimized IR with SW/HW components (memory managers)
Backend
Implementation (SYCL, C, HDL, meta-data, EVEREST APIs)
Standard compilers
Bin/bit-stream
EVEREST Runtime Environment

```
auto A = Matrix(m, n),
     B = Matrix(m, n),
     C = Matrix(m, n);
auto u = Tensor<3>
           (n, n, n);
auto v = (A*B*C)(u);
```

Standard IR format and exchange files

Novel domain-specific extensions (format)

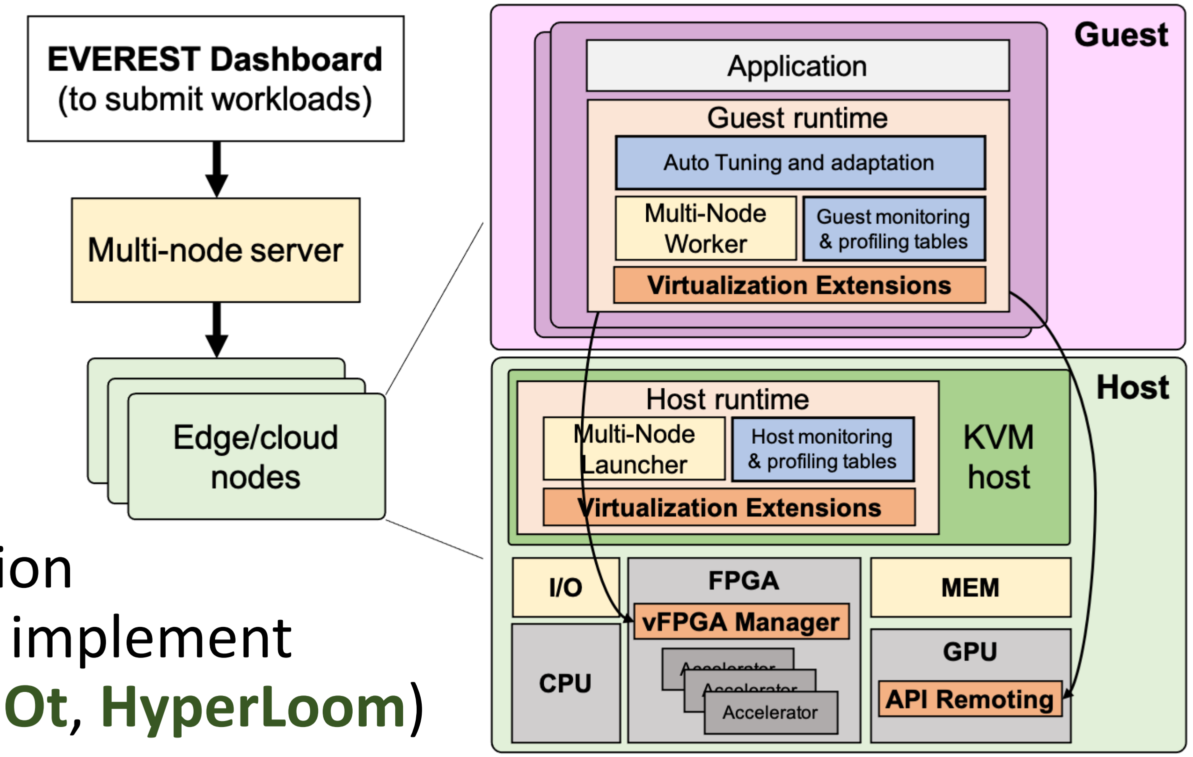
System and resource description (format)

Possibility of using different (ML) frameworks

Interoperability with different HLS tools

EVEREST Programming Environment

2. **Runtime Environment:** implements the *selection of "variants"* and the *hardware configuration* based on the *system status*
   - **Dynamic adaptation and autotuning** (**mARGOt**)
   - **Two-level runtime** for (1) virtualization of hardware resources regardless their distribution and the low-level details of the platforms; (2) implement functional decisions (**VOSYS solutions**, **mARGOt**, **HyperLoom**)



How to collect system status and expose it to the runtime?

Autotuning API

Seamless execution when varying the system configuration (resources, nodes, data, etc.)

Runtime API

Hiding communication latency (e.g., prefetching)

Hardware HPC (Memory) Architectures

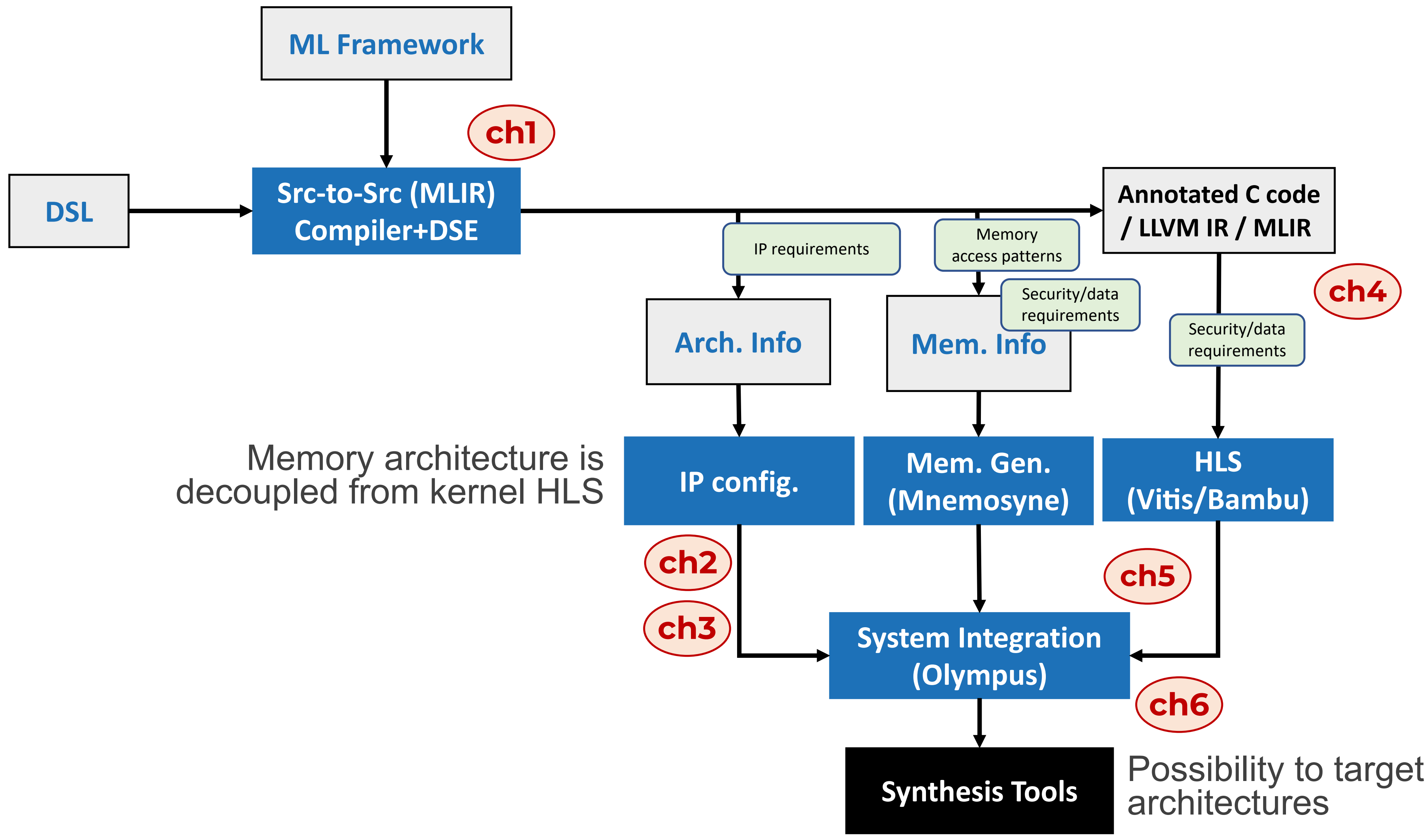
What do we mean with **memory architecture**?

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



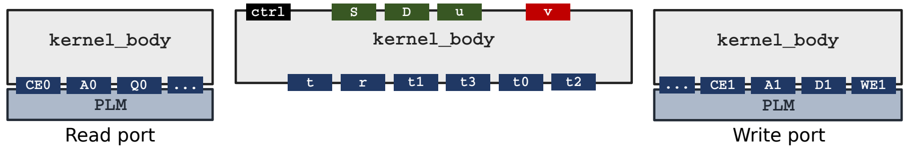
Memory architecture is decoupled from kernel HLS

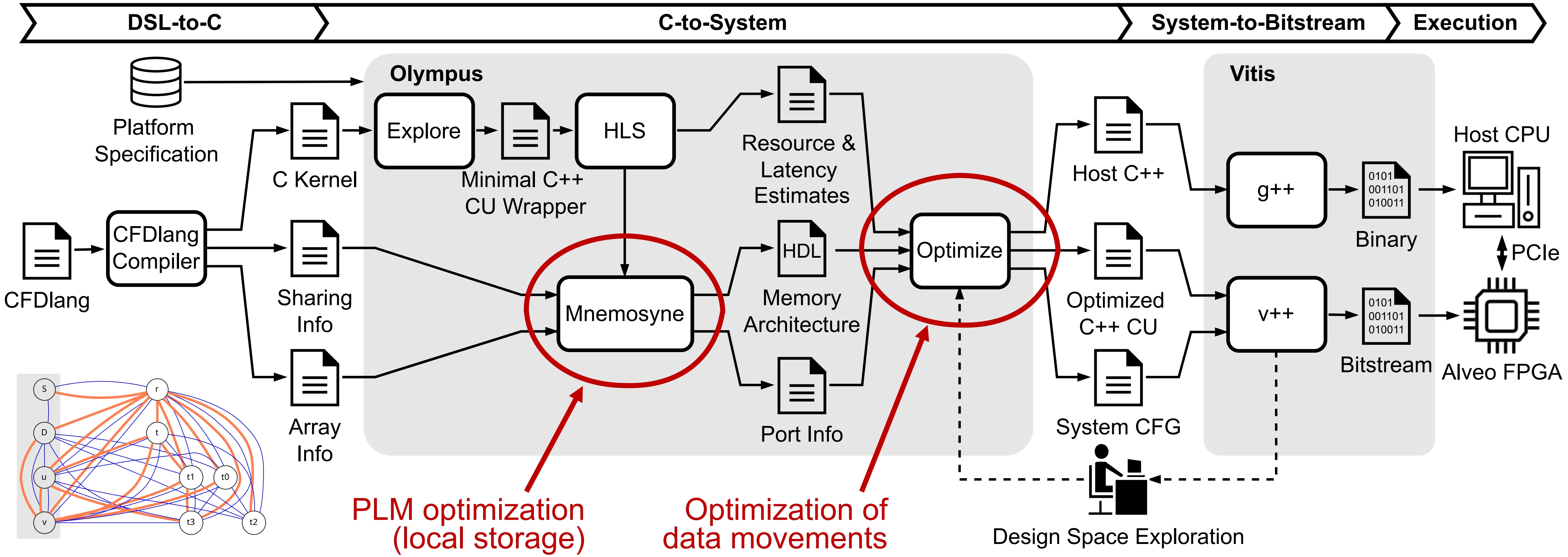
Possibility to target different architectures

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])
```





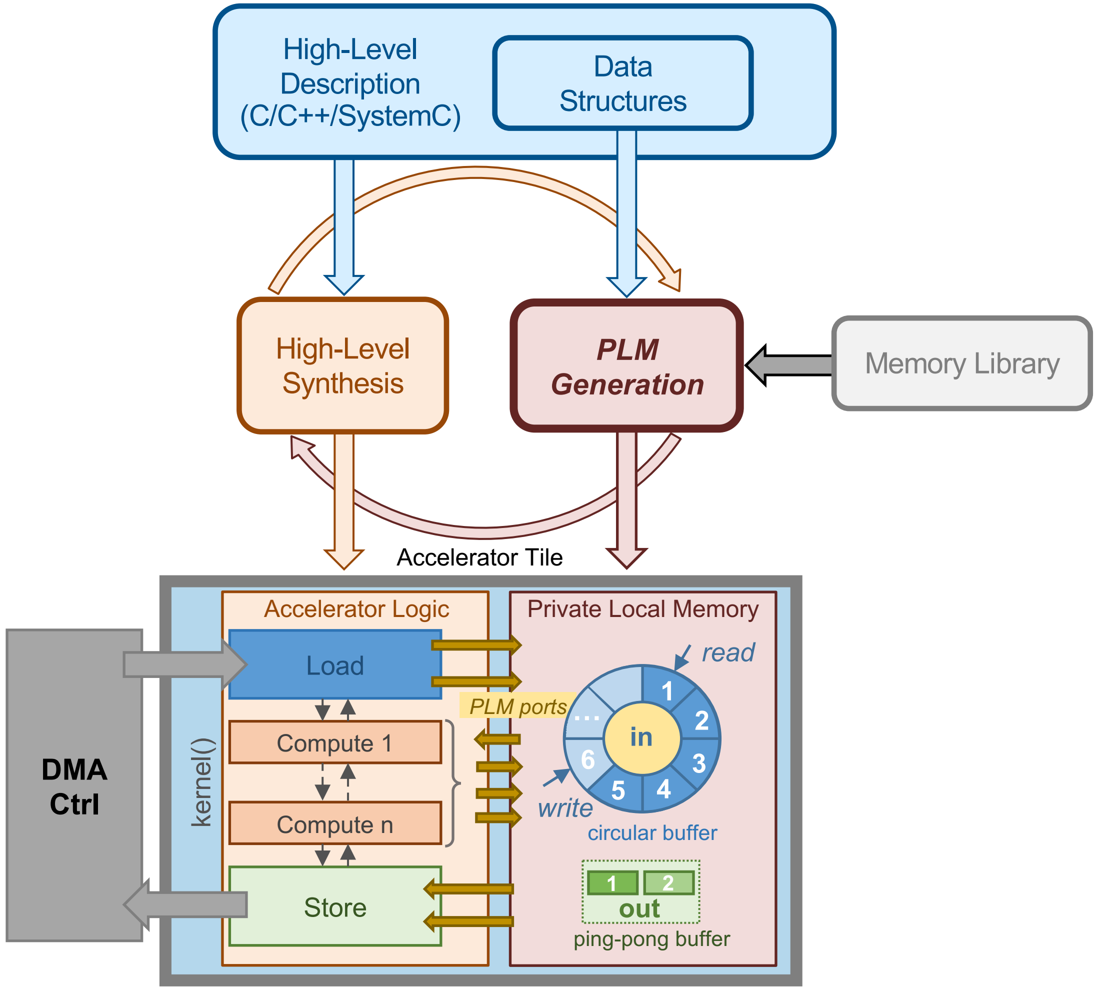
PLM Customization for Heterogeneous SoCs

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

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

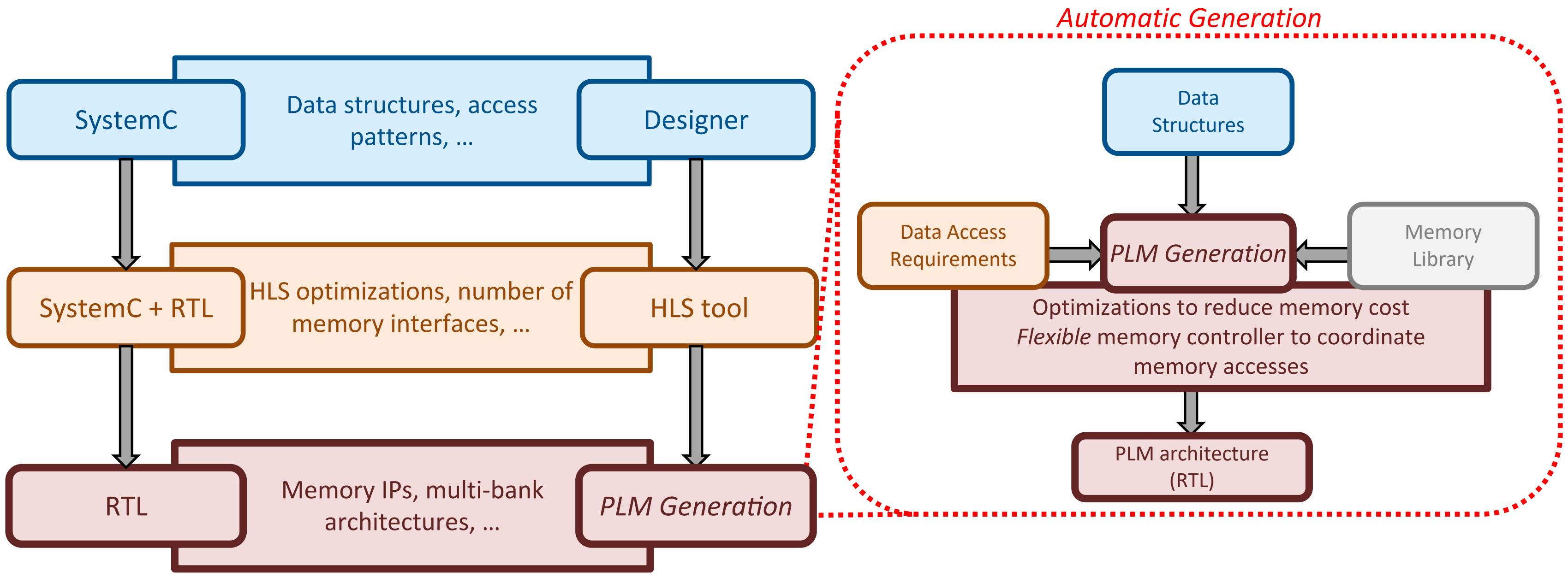
Generation of **specialized PLMs**

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



PLM Customization

System-level methodology for **PLM customization**


Automatic Generation
SystemC
Data structures, access patterns, …
Designer
SystemC + RTL
HLS optimizations, number of memory interfaces, …
HLS tool
RTL
Memory IPs, multi-bank architectures, …
PLM Generation
Data Structures
Data Access Requirements
PLM Generation
Memory Library
Optimizations to reduce memory cost
Flexible memory controller to coordinate memory accesses
PLM architecture (RTL)

Performance optimization: *HLS* defines how the accelerator logic accesses the data structures (e.g., number of parallel accesses)

Cost optimization: *PLM Customization* defines the best PLM microarchitecture to achieve the desired performance (e.g., number of banks, data allocation)

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)"

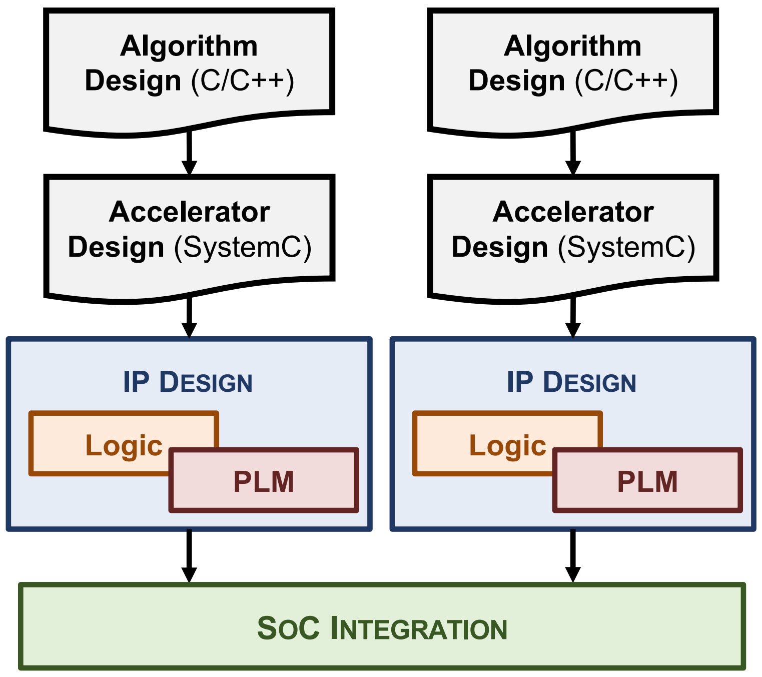
A common case: accelerator kernels never executed at the same time

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

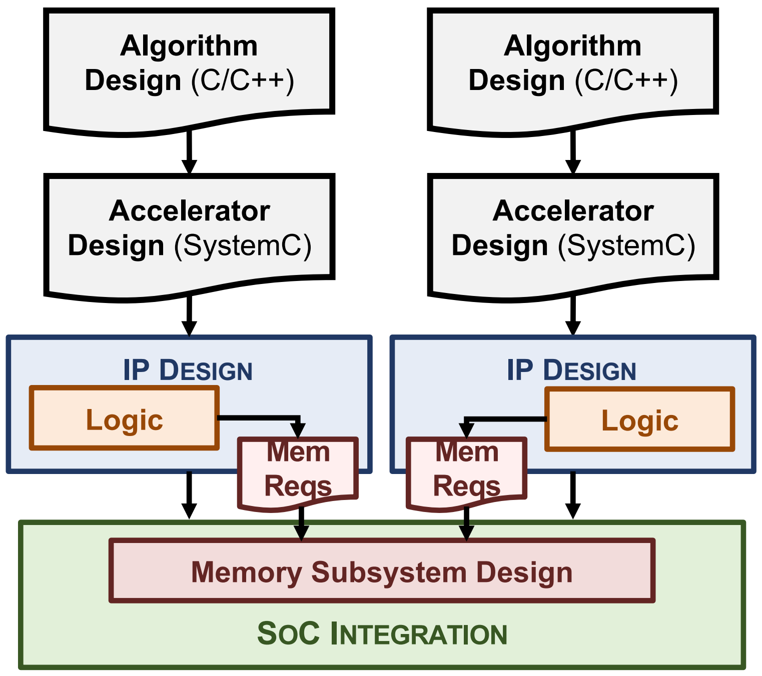
Optimization only at the System-Level

Accelerator(s) memory subsystem is defined during SoC integration

- Possibility for more optimizations



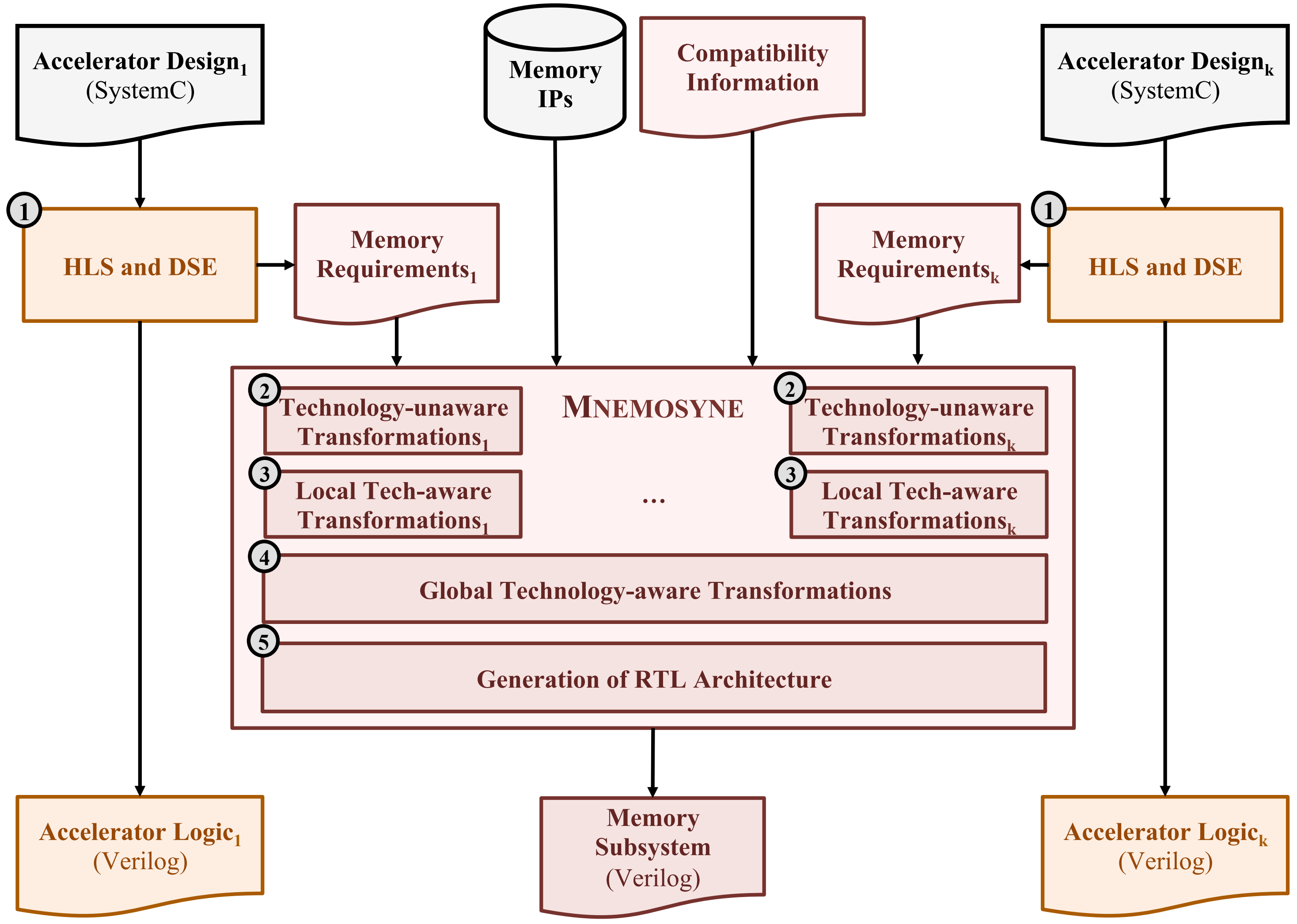
Component-based Approach



System-Level Approach

C. Pilato, L. Carloni, et al. "System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip" TCAD'17

PLM Optimization for Multiple Accelerators

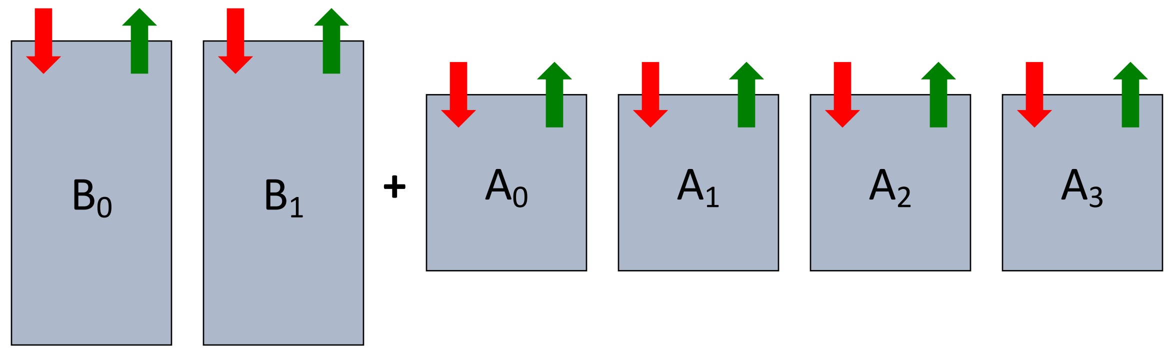


Accelerator Design$_1$ (SystemC)
Memory IPs
Compatibility Information
Accelerator Design$_k$ (SystemC)
1
HLS and DSE
Memory Requirements$_1$
Memory Requirements$_k$
1
HLS and DSE
MNEMOSYNE
2
Technology-unaware Transformations$_1$
2
Technology-unaware Transformations$_k$
3
Local Tech-aware Transformations$_1$
...
3
Local Tech-aware Transformations$_k$
4
Global Technology-aware Transformations
5
Generation of RTL Architecture
Accelerator Logic$_1$ (Verilog)
Memory Subsystem (Verilog)
Accelerator Logic$_k$ (Verilog)

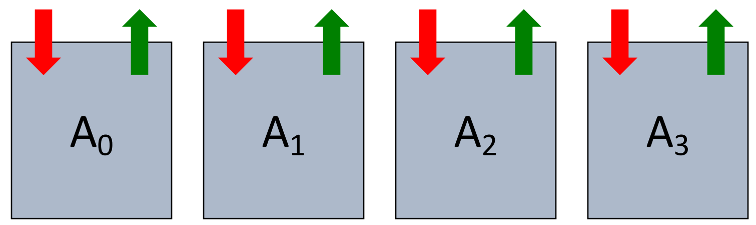
Address-Space Compatibility

Let us assume to have the two following data structures that are never *alive* at the same time

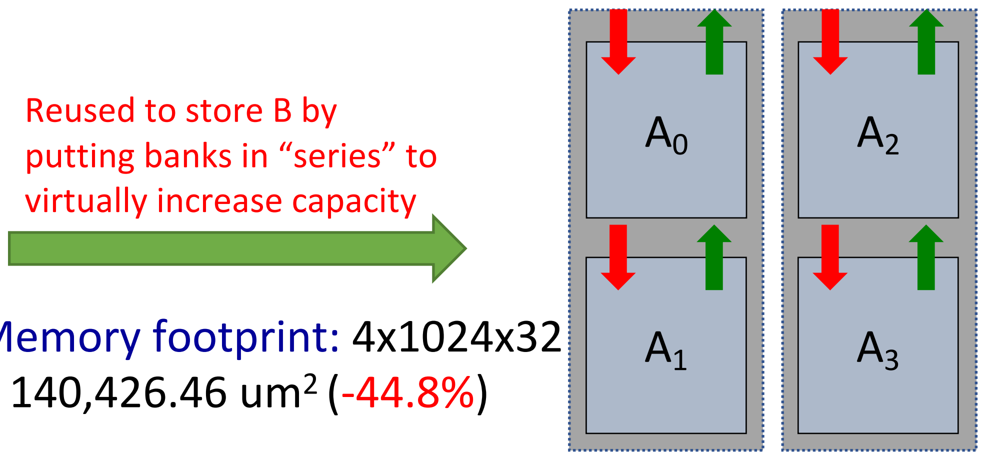
- A[1024] with *data duplication* over 4 parallel banks
- B[4096] with *data distribution* over 2 parallel banks



Memory footprint: 4x1024x32 + 2x2048x32 = 254,485.68 um^2



Reused to store B by putting banks in “series” to virtually increase capacity

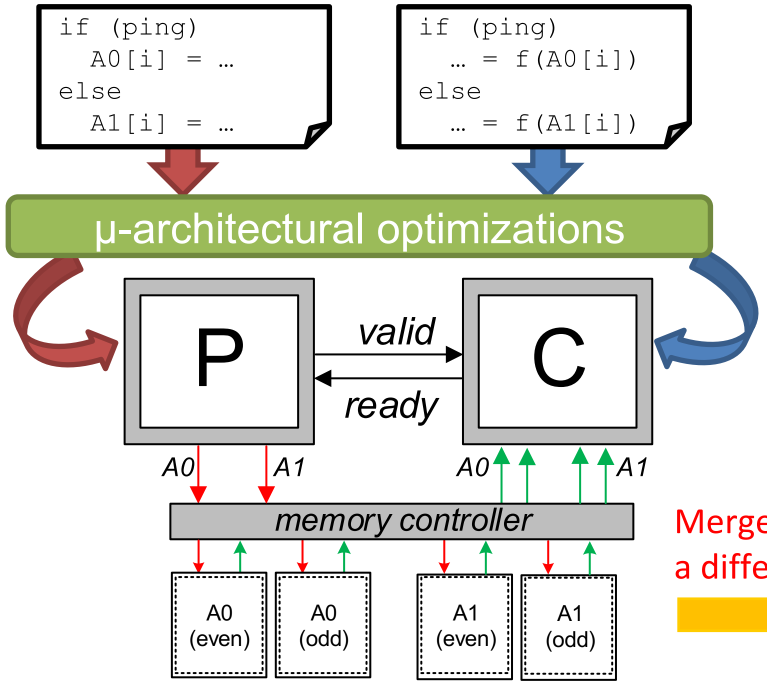


Memory footprint: 4x1024x32 = 140,426.46 um^2 (-44.8%)

Memory-Interface Compatibility

A classical example is the ping-pong buffer (two 2048x16 arrays – A0/A1)

- When process P writes A0 (A1), it never writes A1 (A0)
- When process C reads from A0 (A1), it never reads from A1 (A0)



```
if (ping)
  A0[i] = …
else
  A1[i] = …
```

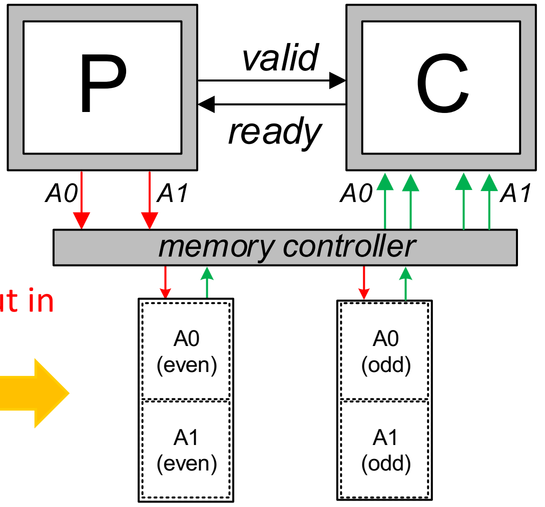
```
if (ping)
  … = f(A0[i])
else
  … = f(A1[i])
```

Memory footprint: 4x1024x32 = 140,426 um^2

Merged in the same IP, but in a different memory space

Memory footprint: 2x2048x32 = 114,059.2 um^2

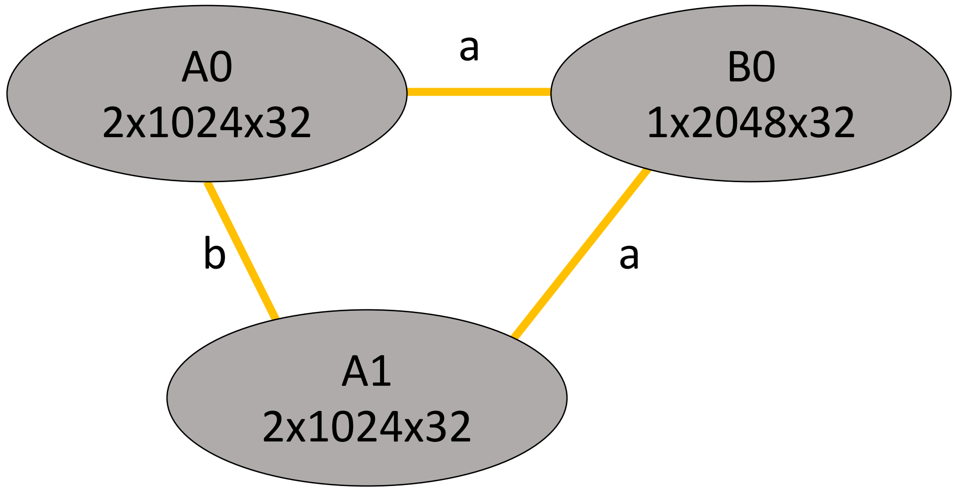
Area reduced by 18% without any performance overhead!



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"

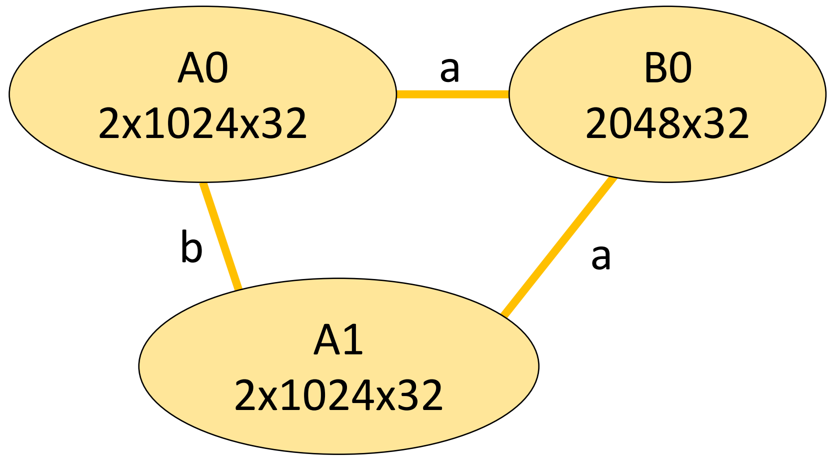


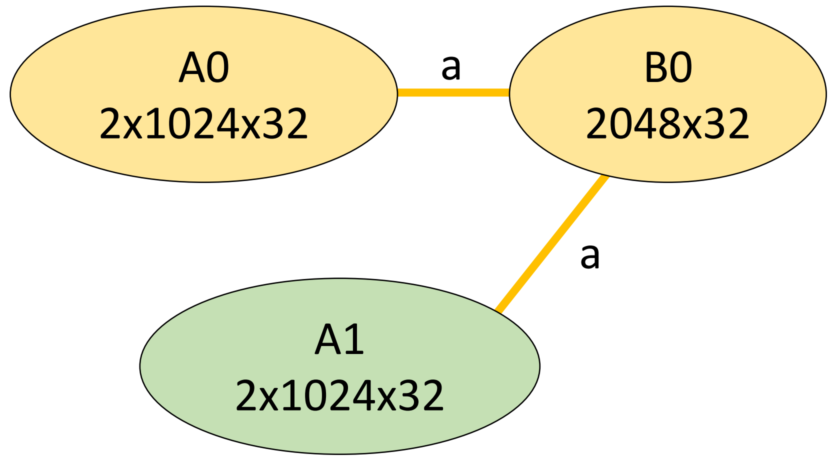
a) **Address-space compatibility**: the data structures are compatible and can use the same memory IPs
b) **Memory-interface compatibility**: the ports are never accessed at the same time and the data structures can stay in the same memory IP

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}?

How to Determine the Memory Subsystem

Clique Enumeration

To define the list of admissible cliques in the MCG

Clique Characterization

To determine the memory architecture of all cliques and their memory cost

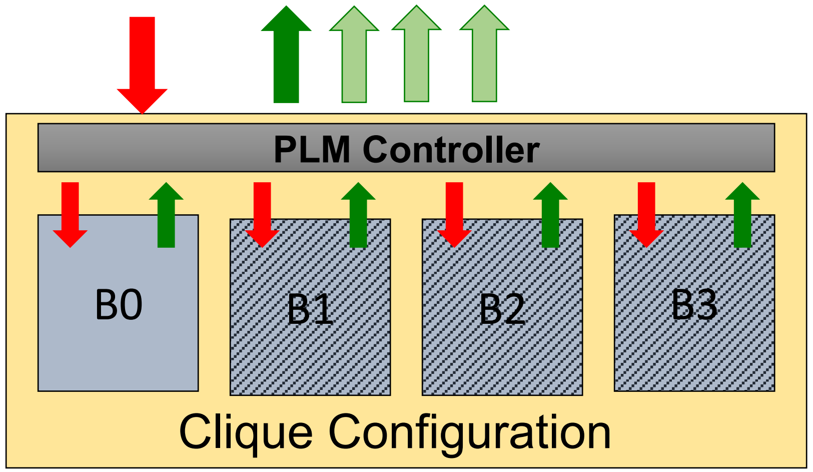
Memory Cost Minimization

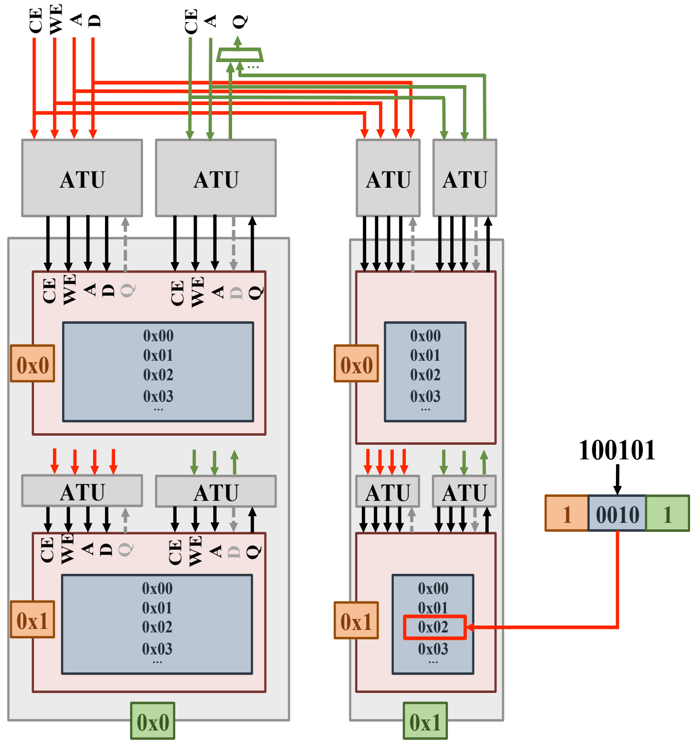
To determine how to partition the MCG such that the total memory cost is minimized

PLM Controller Generation

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

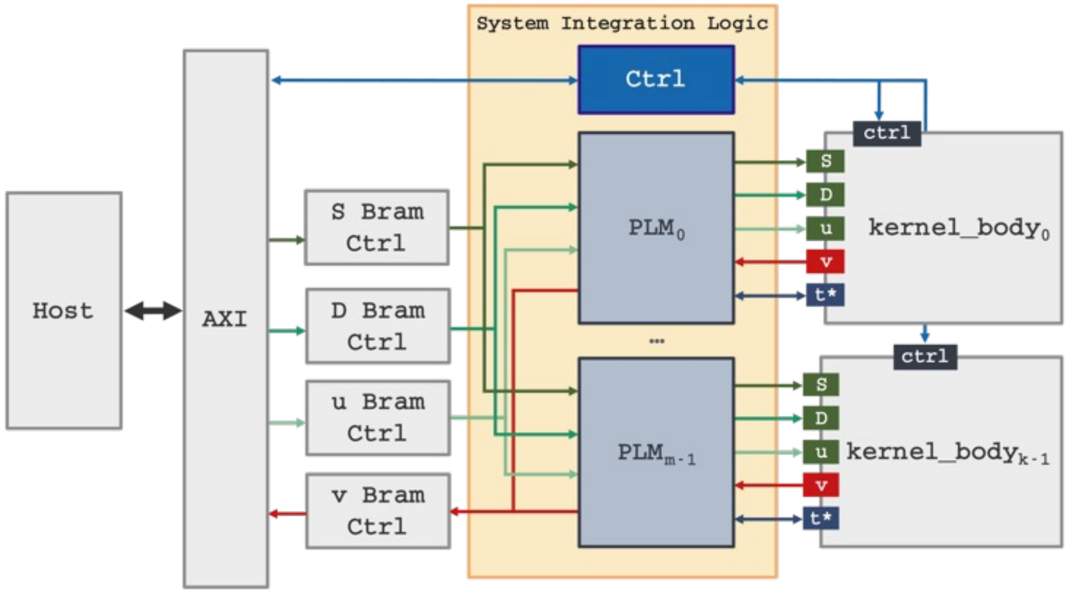
- Accelerator logic is not aware of the actual memory organization
- Array offsets need to be translated into proper memory addresses

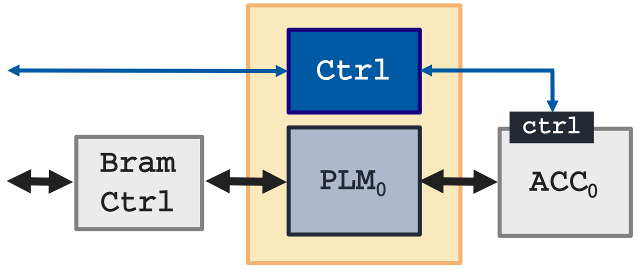


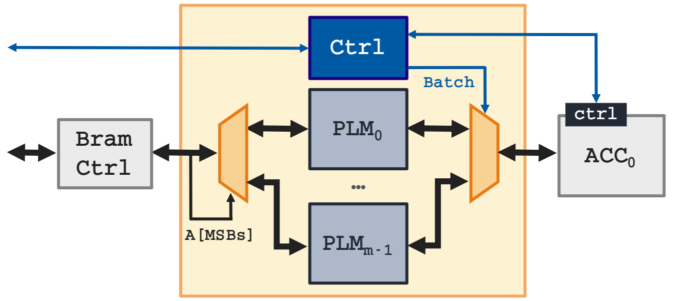


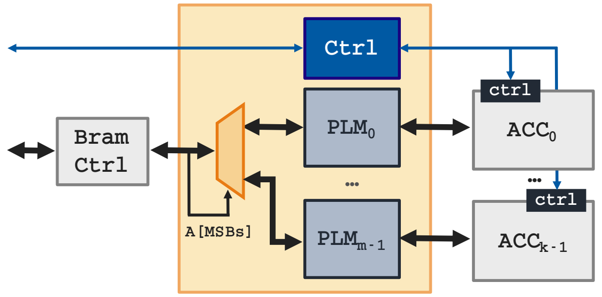
Custom logic with negligible overhead, especially when the number of banks and their size is a power of two

Creation of Parallel Architectures







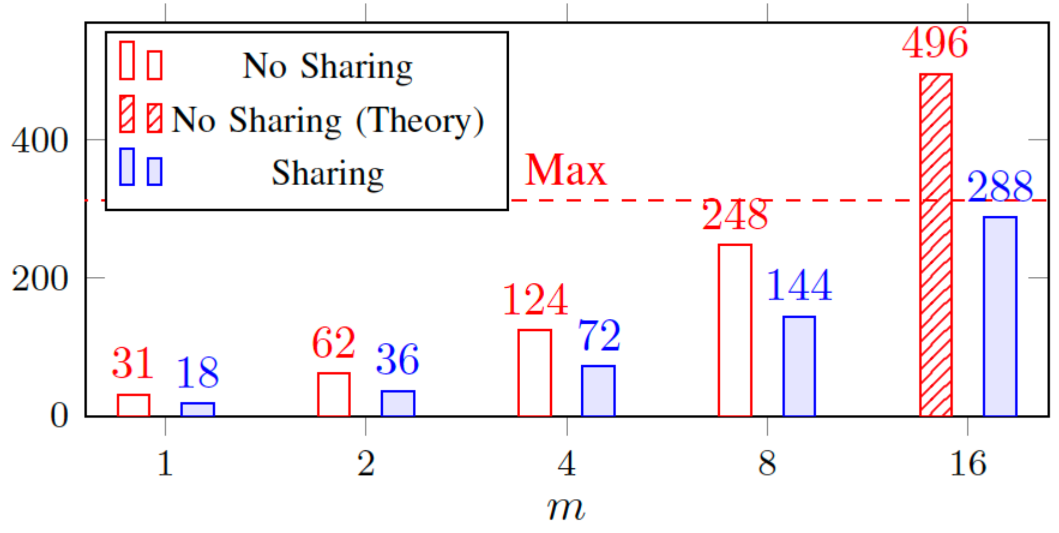


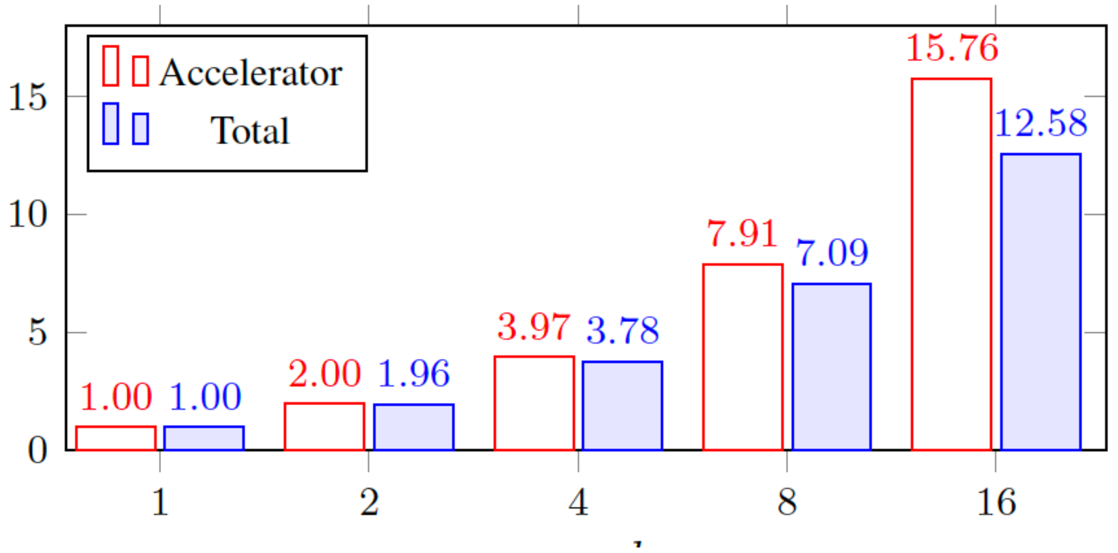
K. F. A. Friebel, S. Soldavini, G. Hempel, C. Pilato, J. Castrillon. "From Domain-Specific Languages to Memory-Optimized Accelerators for Fluid Dynamics" HPCFPGA'21

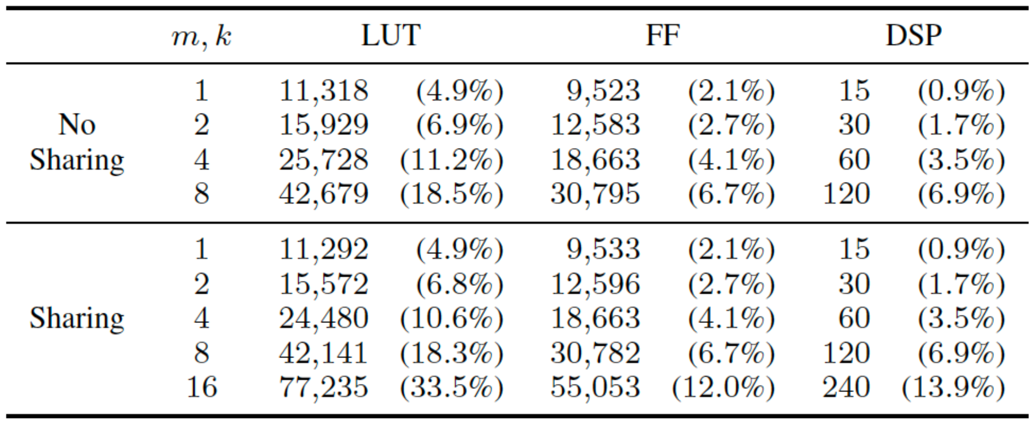
Preliminary Evaluation

- Xilinx Zynq UltraScale+ MPSoC ZCU106 board
  - CFD simulation of 50,000 elements

- Memory sharing allows us to fit more kernels





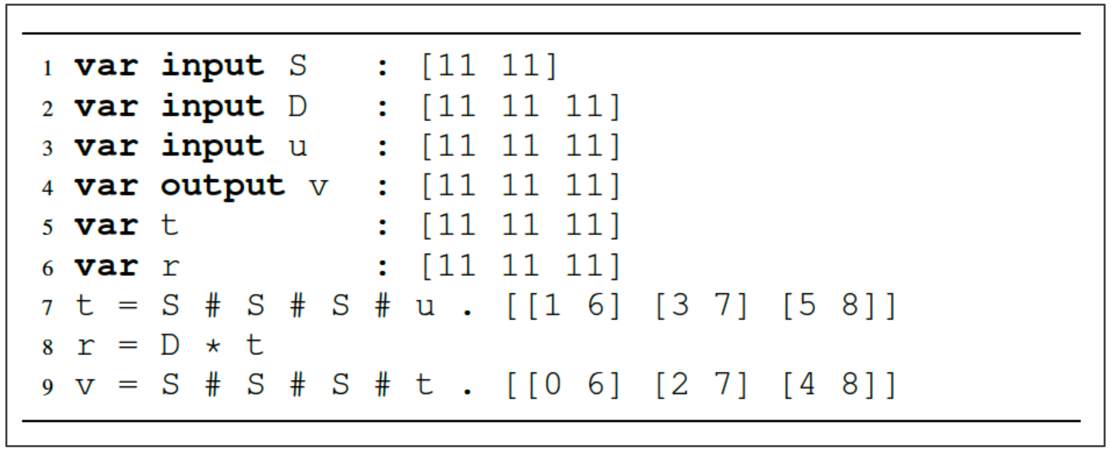


	m, k	LUT		FF		DSP	
No Sharing	1	11,318	(4.9%)	9,523	(2.1%)	15	(0.9%)
	2	15,929	(6.9%)	12,583	(2.7%)	30	(1.7%)
	4	25,728	(11.2%)	18,663	(4.1%)	60	(3.5%)
	8	42,679	(18.5%)	30,795	(6.7%)	120	(6.9%)
Sharing	1	11,292	(4.9%)	9,533	(2.1%)	15	(0.9%)
	2	15,572	(6.8%)	12,596	(2.7%)	30	(1.7%)
	4	24,480	(10.6%)	18,663	(4.1%)	60	(3.5%)
	8	42,141	(18.3%)	30,782	(6.7%)	120	(6.9%)
	16	77,235	(33.5%)	55,053	(12.0%)	240	(13.9%)

POLITECNICO MILANO 1863

Next Step: System-Level DSL

CFDlang: DSL for representing the kernel



```
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]]
```

Moving to a system-level representation

- Simple example for a massively parallel architecture:

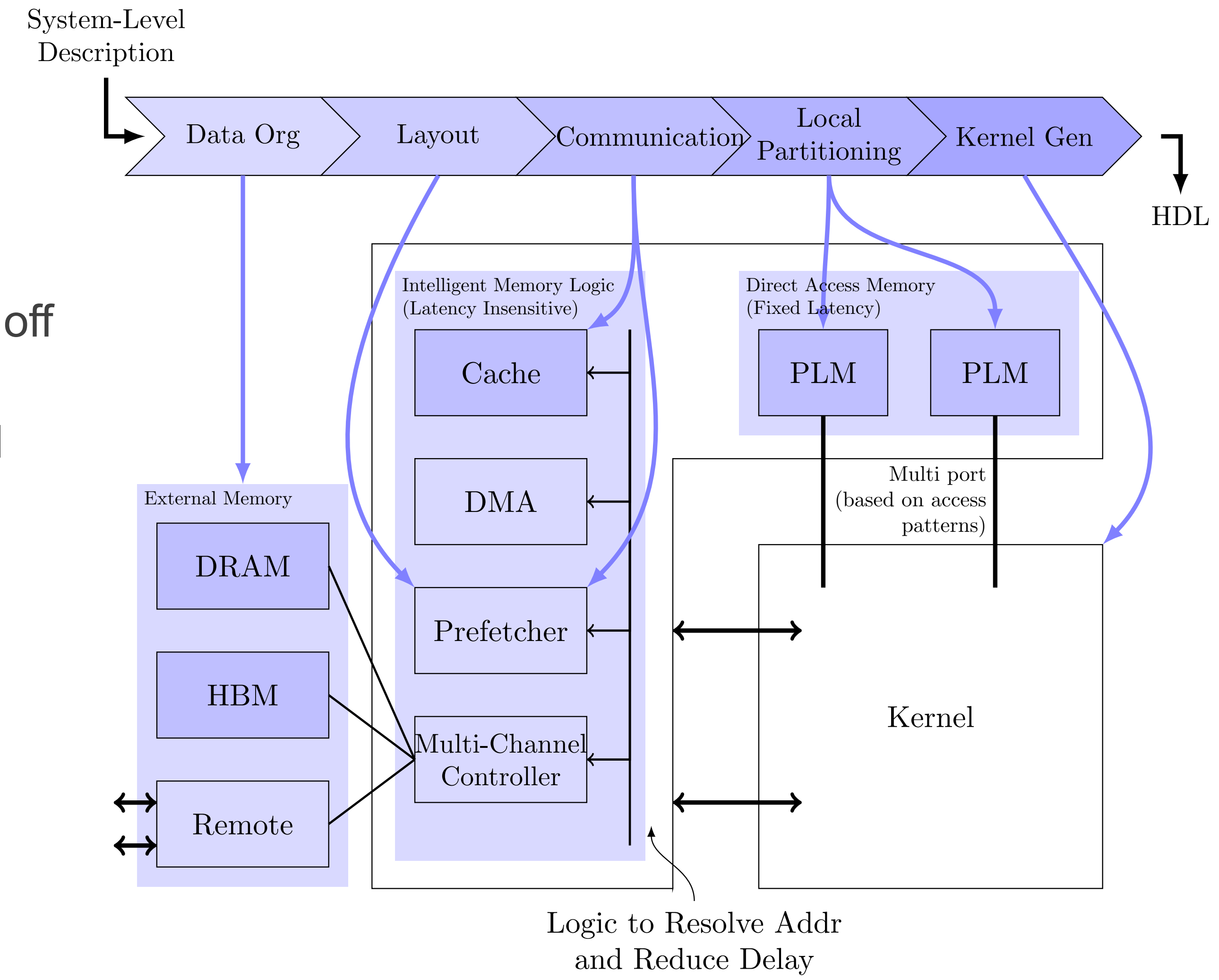
LOOP ~ KERNEL(S, D, u, v)

Possibility to decide the memory layout and configure DMA/prefetchers based on the target architecture/platform

Next Step: Let's Put Memory First

We are building an MLIR **compilation flow** for **automatic memory specialization**:

- **MLIR Input** – DSL description of the system functionality
- **Data Organization** – Determine which data resides off chip (also based on user/compiler annotations)
- **Layout** – Reorganize communication to exploit local memories and perform efficient parallel computation
- **Communication** – Configure prefetcher to hide transfer latency
- **Local Partitioning** – Determine multi-bank PLM architecture (Mnemosyne)
- **HLS** – Generate computation part (interfacing with existing HLS tools, e.g., open-source Vitis HLS frontend)
- **HDL Output** – Automated code generation and system-level integration based on the target platform



S. Soldavini and C. Pilato. "Compiler Infrastructure for Specializing Domain-Specific Memory Templates" LATTE'21

Olympus – Automated System-Level Integration

We are developing a complete hardware architecture generation flow based on **MLIR description** of the **system functionality**

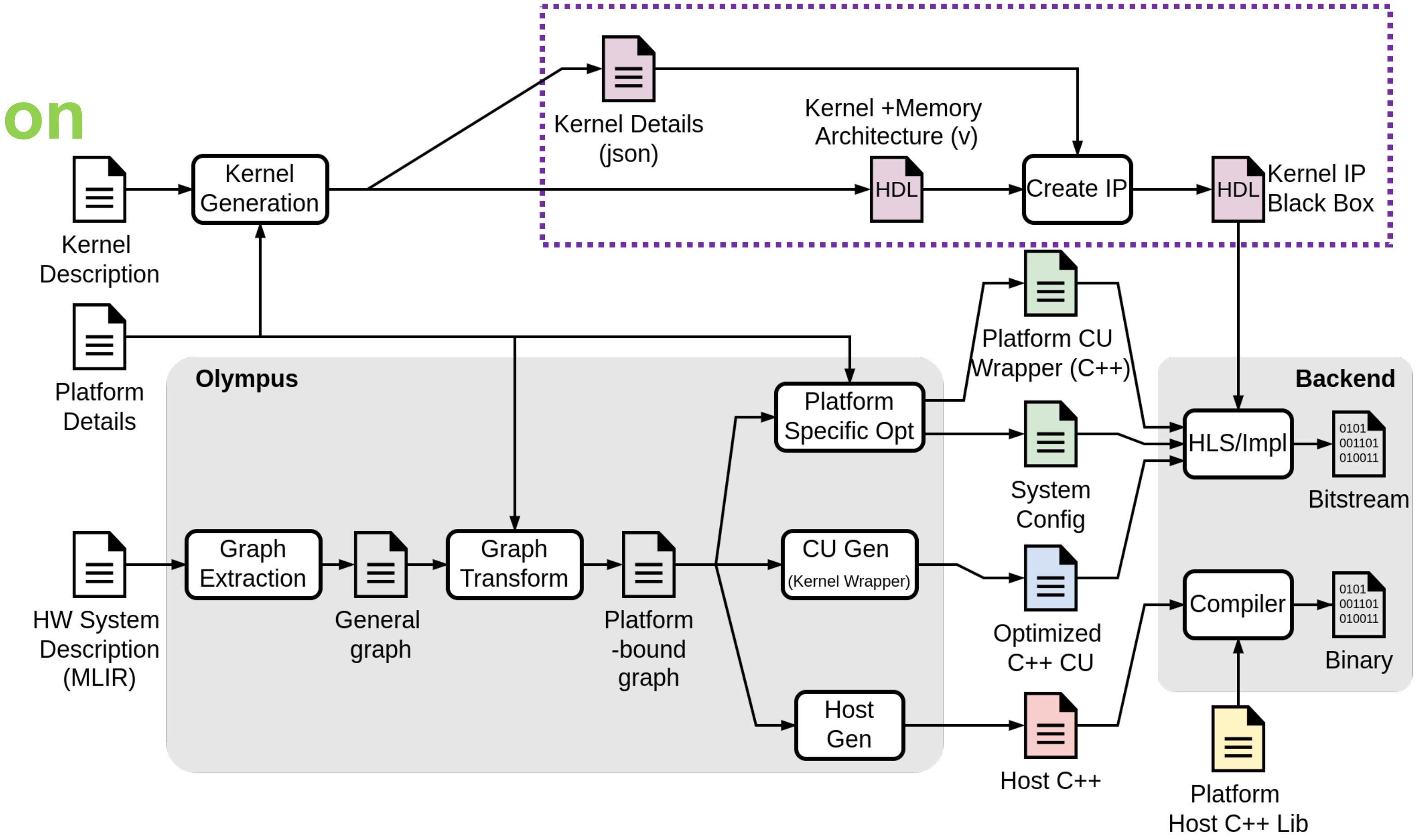
Platform-specific description

- HBM-based Xilinx Alveo
- IBM CloudFPGA
- …

Host code generation

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

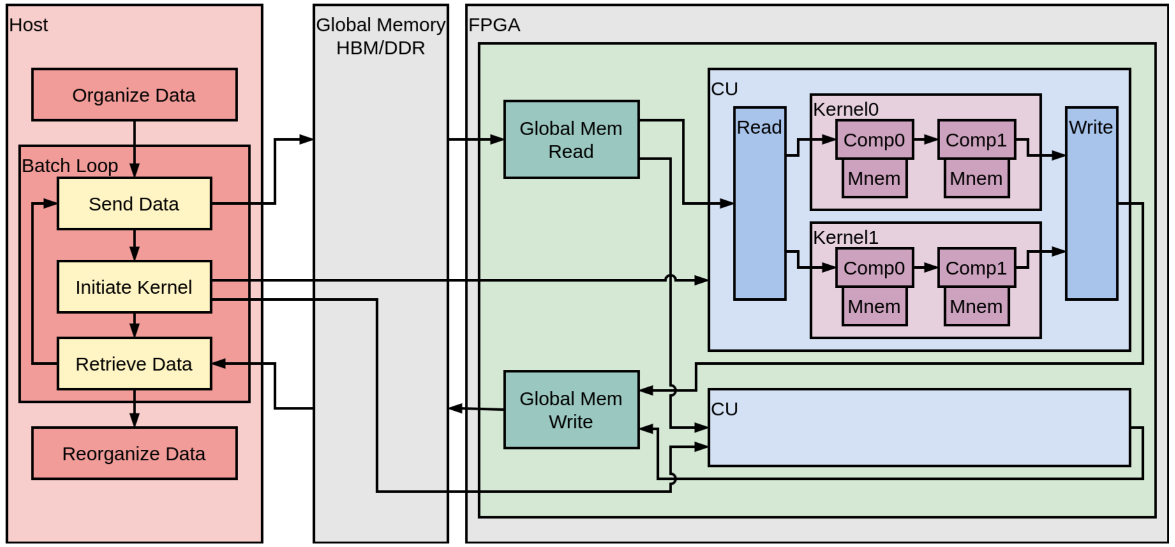
Possibility to use several HLS tools/HDL generators



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

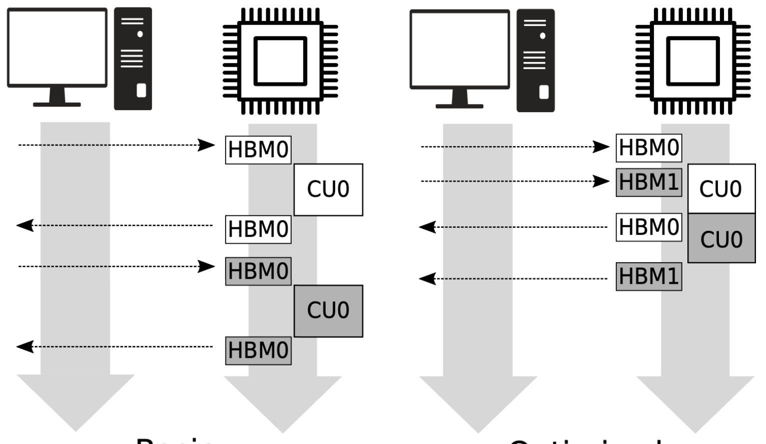
- **Synthesizable C++ code** that includes:
  - **Accelerators** and **PLM** generated with HLS
  - **Communication modules** to match interfaces
    - *Standard AXI interfaces to the system (either cloudFPGA SHELL or HBM channels)*
    - May include **"intelligent" policies** to coordinate (or protect) 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*

POLITECNICO MILANO 1863

From MLIR to System Architecture

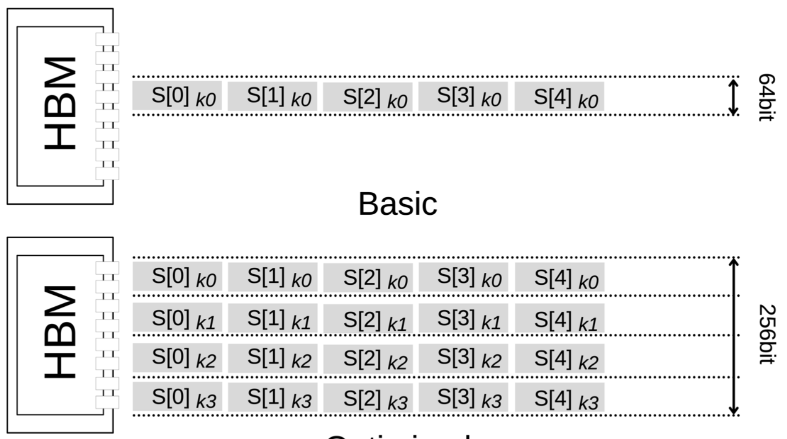
Automatic integration of memory optimizations for **high-performance data transfers**, such as:

- **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



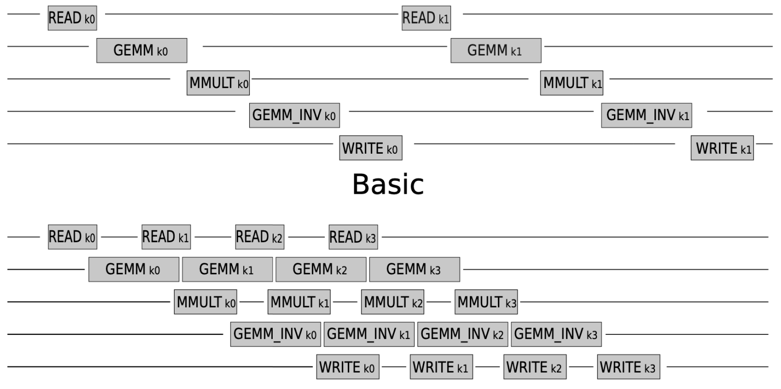
Basic Optimized

Double buffering


HBM
S[0] k0 S[1] k0 S[2] k0 S[3] k0 S[4] k0
64bit
Basic
HBM
S[0] k0 S[1] k0 S[2] k0 S[3] k0 S[4] k0
S[0] k1 S[1] k1 S[2] k1 S[3] k1 S[4] k1
S[0] k2 S[1] k2 S[2] k2 S[3] k2 S[4] k2
S[0] k3 S[1] k3 S[2] k3 S[3] k3 S[4] k3
256bit

Optimized

Bus optimization

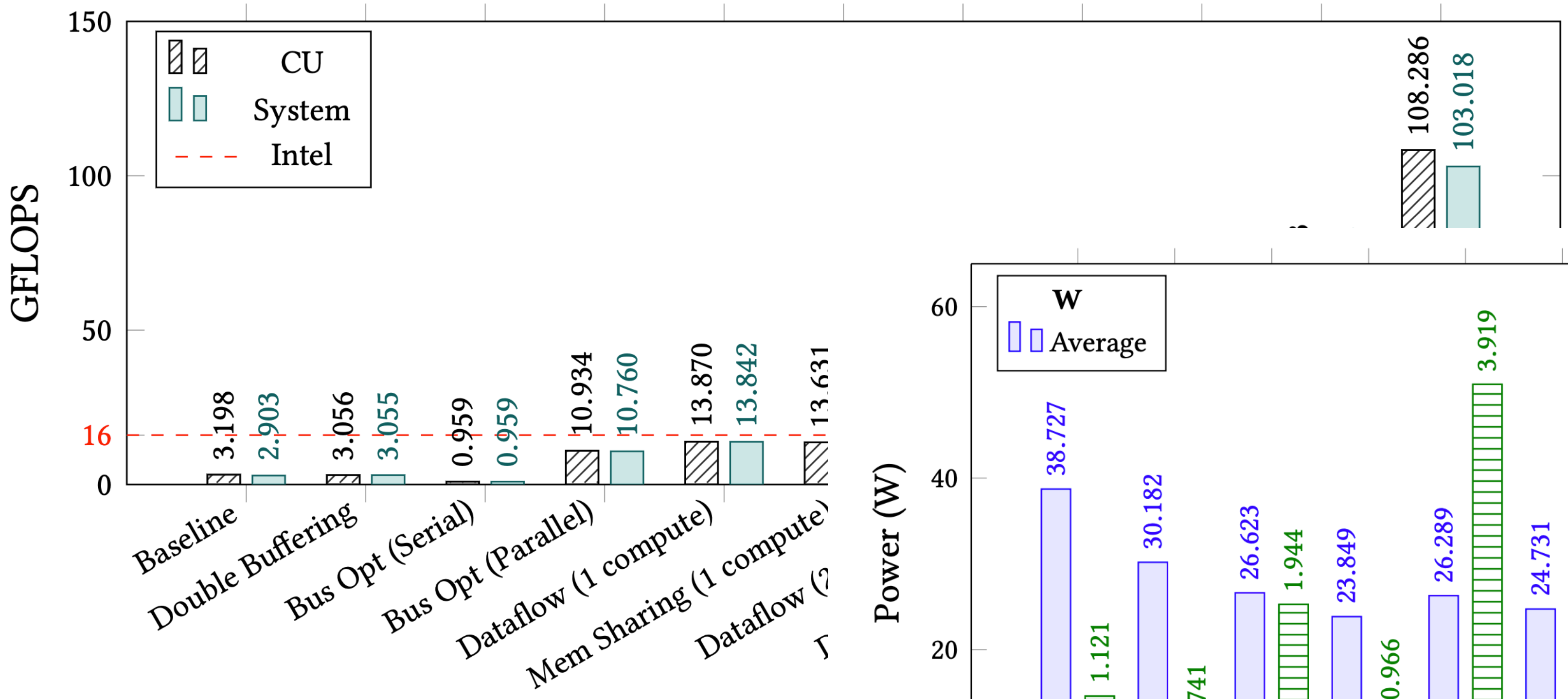


Optimized

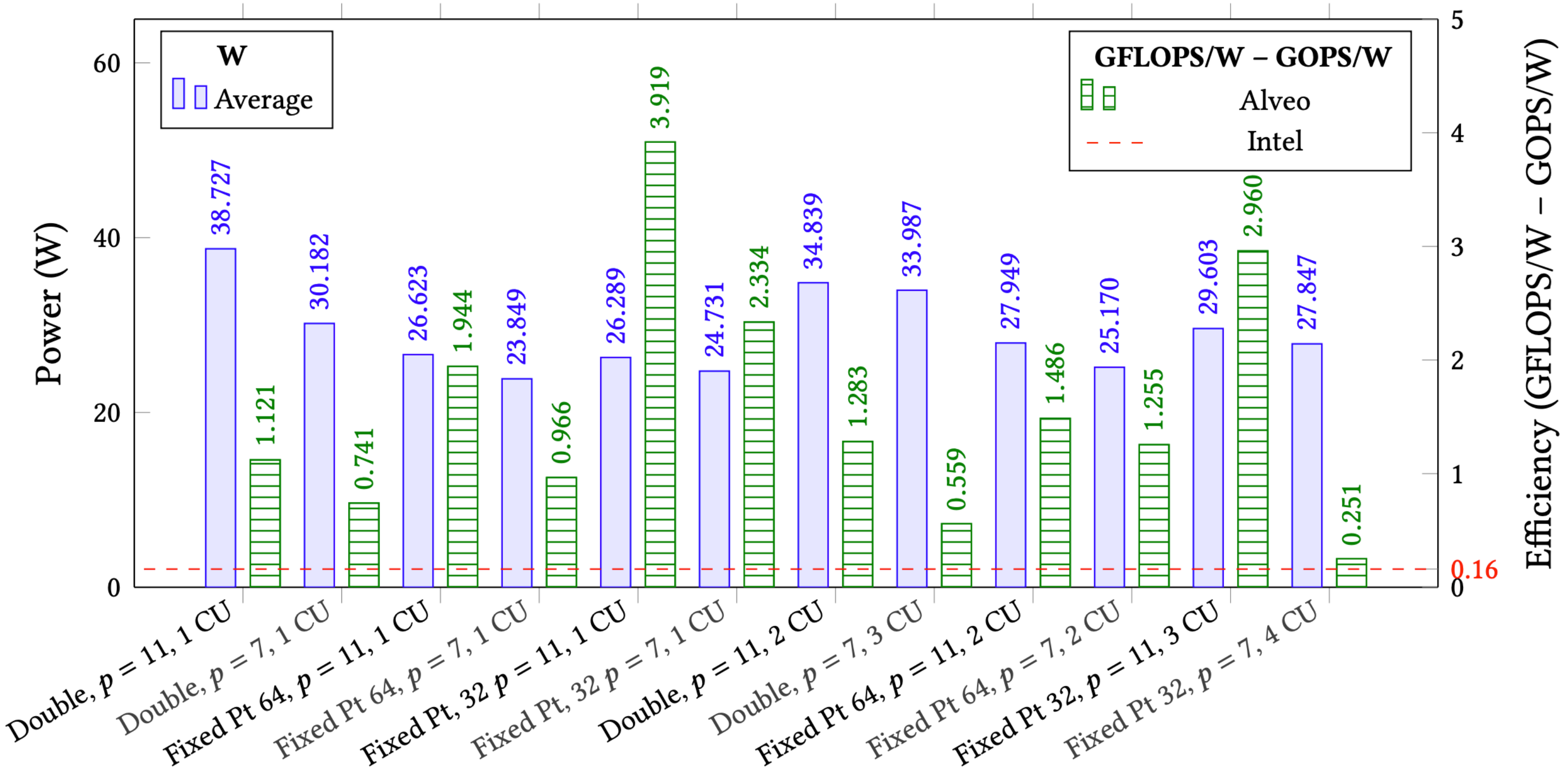
Dataflow

POLITECNICO MILANO 1863

Results on HBM FPGA



Best performance: 103 GOPS
(118x faster than our "starting point")



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" arXiv'22

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**

Work done in collaboration with Stephanie Soldavini (Politecnico di Milano), Mattia Tibaldi (Politecnico di Milano), Jeronimo Castrillon (TU Dresden), Karl F. A. Friebel (TU Dresden), and Gerald Hempel (TU Dresden)

Thank you!

Christian Pilato, christian.pilato@polimi.it

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