Machine Learning Integration in TASTE for FPGA-Based Space Systems

Fabrizio Ferrandi, Michele Fiorito

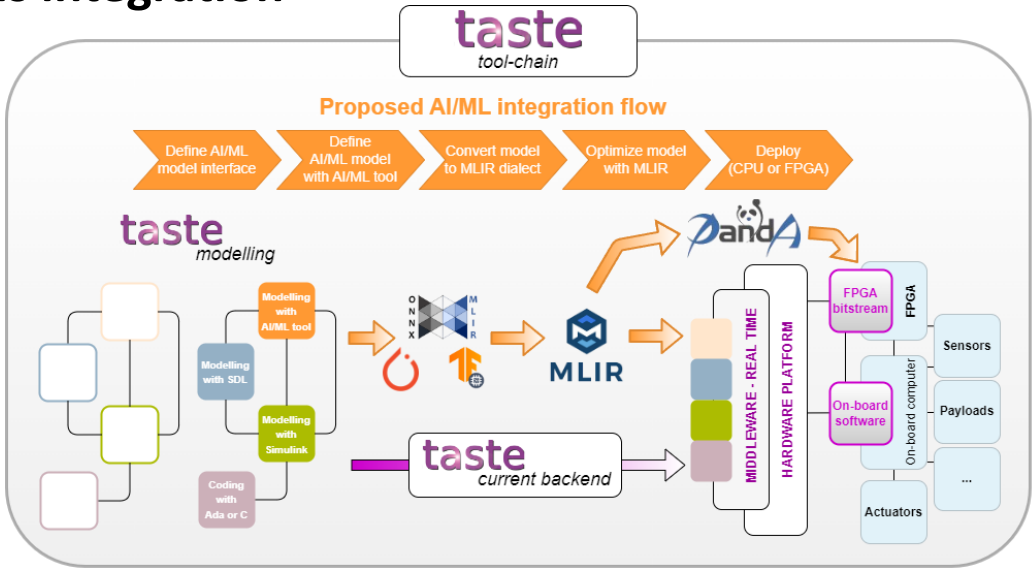
Politecnico di Milano, Italy,

Outline

- **OSIP**
- **AI/ML-based design flow**
- **Memory optimization**
- **Kernel extraction**
- **Co-simulation optimization**
- **Conclusion**

Open Space Innovation Project (OSIP)

- **How does Machine Learning TASTE:**
  - **Extending Model-Based Design techniques to support machine learning applications integration**



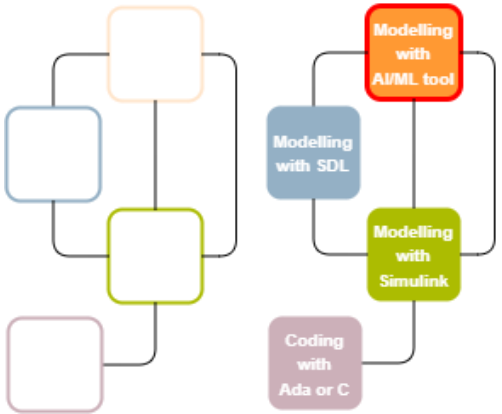
Fabrizio Ferrandi - Michele Fiorito
Politecnico di Milano

TASTE Toolchain Integration

- **Model-based design toolchain for embedded systems**
- **Unified approach to define system functionalities**
- **Automated code generation from DSL (Matlab/SDL) to target hardware (CPU/FPGA)**
- **Extension for AI/ML models support**
- **Target-specific AI/ML TASTE module generation from ONNX description through the proposed MLIR/HLS toolchain**

taste
tool-chain

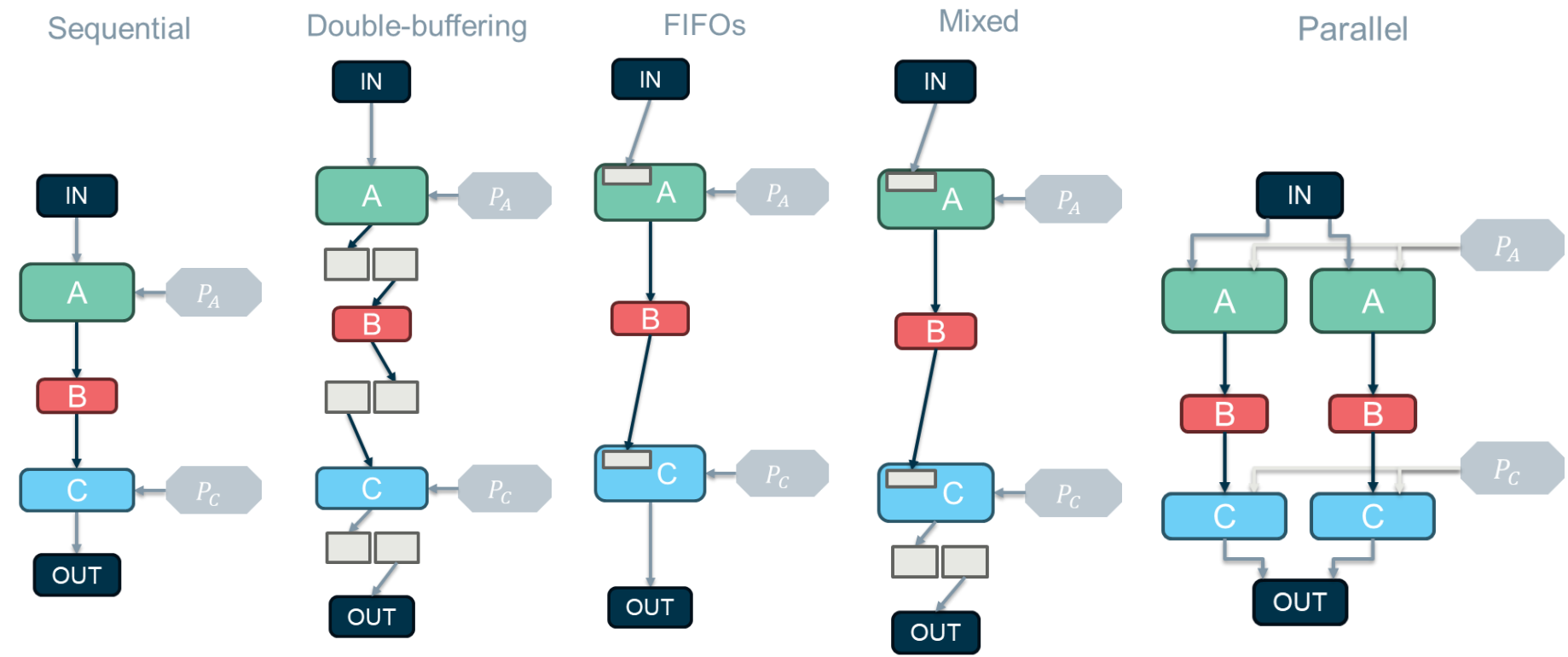
taste
modelling



A ML/AI compilation-based approach

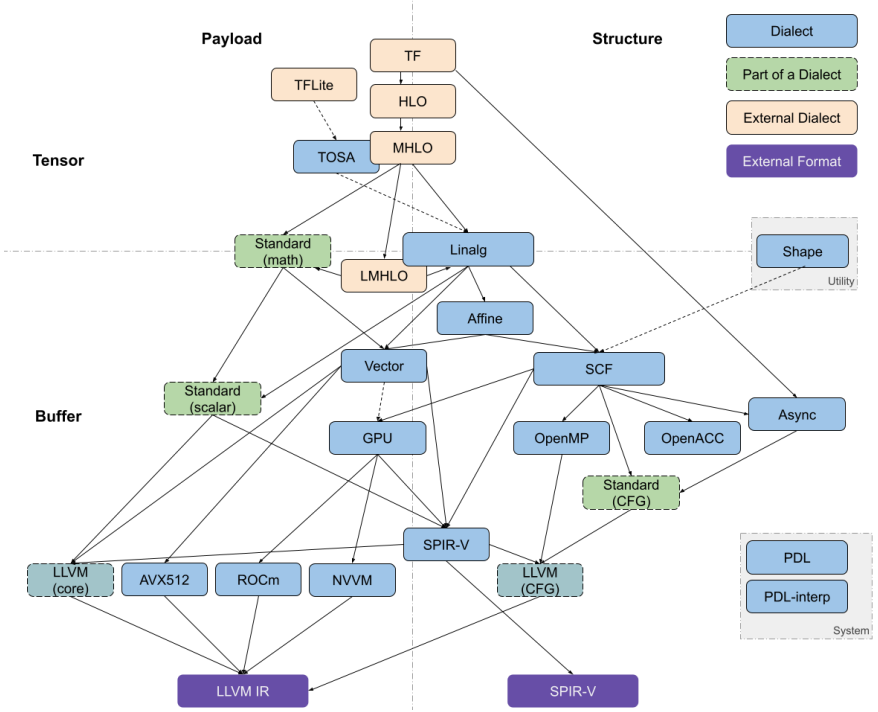
- **Exploit domain-specific tool for the AI/ML model high-level description**
- **Optimize the AI/ML model through MLIR**
  - **Iterative lowering and high-level optimizations**
  - **LLVM IR generation**
- **Synthesize the generated IR through Panda Bambu HLS**
  - **Low-level optimizations (loop pipelining, memory partitioning, parallelization, …)**
  - **RTL generation**

Computational Models: CPUs/FPGAs



Multi-Level IR Compiler Framework

- **Extensible and reusable compiler infrastructure**
- **Supports many different ‘Dialects’ to represent specific data structures**
- **Each dialect exposes a specific feature of the represented data structure to ease its optimization**
- **Interoperability between dialects is possible through ‘lowering’ operations**
- **Code generation is enabled through the lowering to LLVM IR**

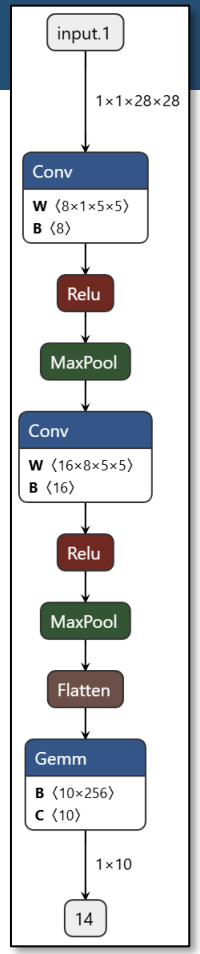


MLIR Optimizations

- ***Linalg/TOSA/Onnx* dialect**
  - **Dataflow optimizations**
  - **Graph parallelization**
- ***Affine* dialect**
  - **Polyhedral transformation to optimize data locality/memory access patterns**
  - **Loop unrolling/tiling/coalescing**
  - **Target primitives mapping (tensor instructions/AI coprocessor)**
- ***Vector* dialect**
  - **IR vectorization**
  - **Target primitives mapping (vector instructions)**
- ***Memref* dialect**
  - **Memory allocation optimization**

Isomorphic Kernel Outlining Pass

- **Optimization step inspired by the Identical Code Folding compiler pass:**
  - ➢ **Identifies and merges identical functions**
- **The idea is to identify recurring layers or operations, merge and generalize them, and optimize them.**



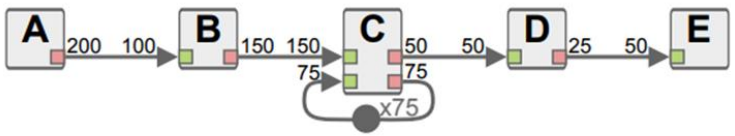
Example ONNX Model

```
func.func @main_graph(%arg0: tensor<1x1x28x28xf32>) -> (tensor<1x10xf32>) {
  %0 = onnx.Constant dense<__elided__> : tensor<8x1x5x5xf32>
  %1 = onnx.Constant dense<__elided__> : tensor<8xf32>
  %2 = onnx.Constant dense<__elided__> : tensor<16x8x5x5xf32>
  %3 = onnx.Constant dense<__elided__> : tensor<16xf32>
  %4 = onnx.Constant dense<__elided__> : tensor<10x256xf32>
  %5 = onnx.Constant dense<__elided__> : tensor<10xf32>
  %6 = "onnx.Conv"(%arg0, %0, %1) : (tensor<1x1x28x28xf32>, tensor<8x1x5x5xf32>, tensor<8xf32>) -> tensor<1x8x24x24xf32>
  %7 = "onnx.Relu"(%6) : (tensor<1x8x24x24xf32>) -> tensor<1x8x24x24xf32>
  %8 = "onnx.MaxPoolSingleOut"(%7) : (tensor<1x8x24x24xf32>) -> tensor<1x8x12x12xf32>
  %9 = "onnx.Conv"(%8, %2, %3) : (tensor<1x8x12x12xf32>, tensor<16x8x5x5xf32>, tensor<16xf32>) -> tensor<1x16x8x8xf32>
  %10 = "onnx.Relu"(%9) : (tensor<1x16x8x8xf32>) -> tensor<1x16x8x8xf32>
  %11 = "onnx.MaxPoolSingleOut"(%10) : (tensor<1x16x8x8xf32>) -> tensor<1x16x4x4xf32>
  %12 = "onnx.Flatten"(%11) : (tensor<1x16x4x4xf32>) -> tensor<1x256xf32>
  %13 = "onnx.Gemm"(%12, %4, %5) : (tensor<1x256xf32>, tensor<10x256xf32>, tensor<10xf32>) -> tensor<1x10xf32>
  return %13 : tensor<1x10xf32>
}
```

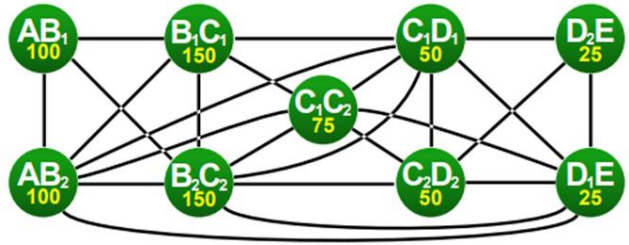
Kernel Outlining Example

```
func.func @mnist_float32_cap_kernel(
       %arg0: tensor<?x?x?x?xf32>, %arg1: tensor<?x?x?x?xf32>, %arg2: tensor<?xf32>) -> tensor<?x?x?x?xf32>
{
  %0 = "onnx.Conv"(%arg0, %arg1, %arg2) : (tensor<?x?x?x?xf32>, tensor<?x?x?x?xf32>, tensor<?xf32>) -> tensor<?x?x?x?xf32>
  %1 = "onnx.Relu"(%0) : (tensor<?x?x?x?xf32>) -> tensor<?x?x?x?xf32>
  %2 = "onnx.MaxPoolSingleOut"(%1) : (tensor<?x?x?x?xf32>) -> tensor<?x?x?x?xf32>
  return %2 : tensor<?x?x?x?xf32>
}
func.func @main_graph(%arg0: tensor<1x1x28x28xf32>) -> (tensor<1x10xf32>) {
  ...
  %6 = call @mnist_float32_cap_kernel(%cast, %cast_0, %cast_1) :
             (tensor<?x?x?x?xf32>, tensor<?x?x?x?xf32>, tensor<?xf32>) -> tensor<?x?x?x?xf32>
  ...
  %8 = call @mnist_float32_cap_kernel(%cast_3, %cast_4, %cast_5) :
             (tensor<?x?x?x?xf32>, tensor<?x?x?x?xf32>, tensor<?xf32>) -> tensor<?x?x?x?xf32>
  ...
  %10 = "onnx.Flatten"(%9) : (tensor<1x16x4x4xf32>) -> tensor<1x256xf32>
  %11 = "onnx.Gemm"(%10, %4, %5) : (tensor<1x256xf32>, tensor<10x256xf32>, tensor<10xf32>) -> tensor<1x10xf32>
  return %11 : tensor<1x10xf32>
}
```

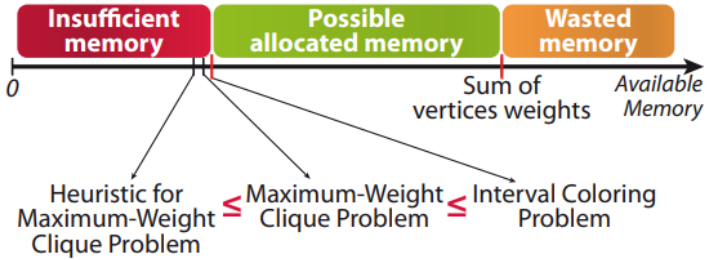
Memory Analysis



Synchronous Dataflow Graph

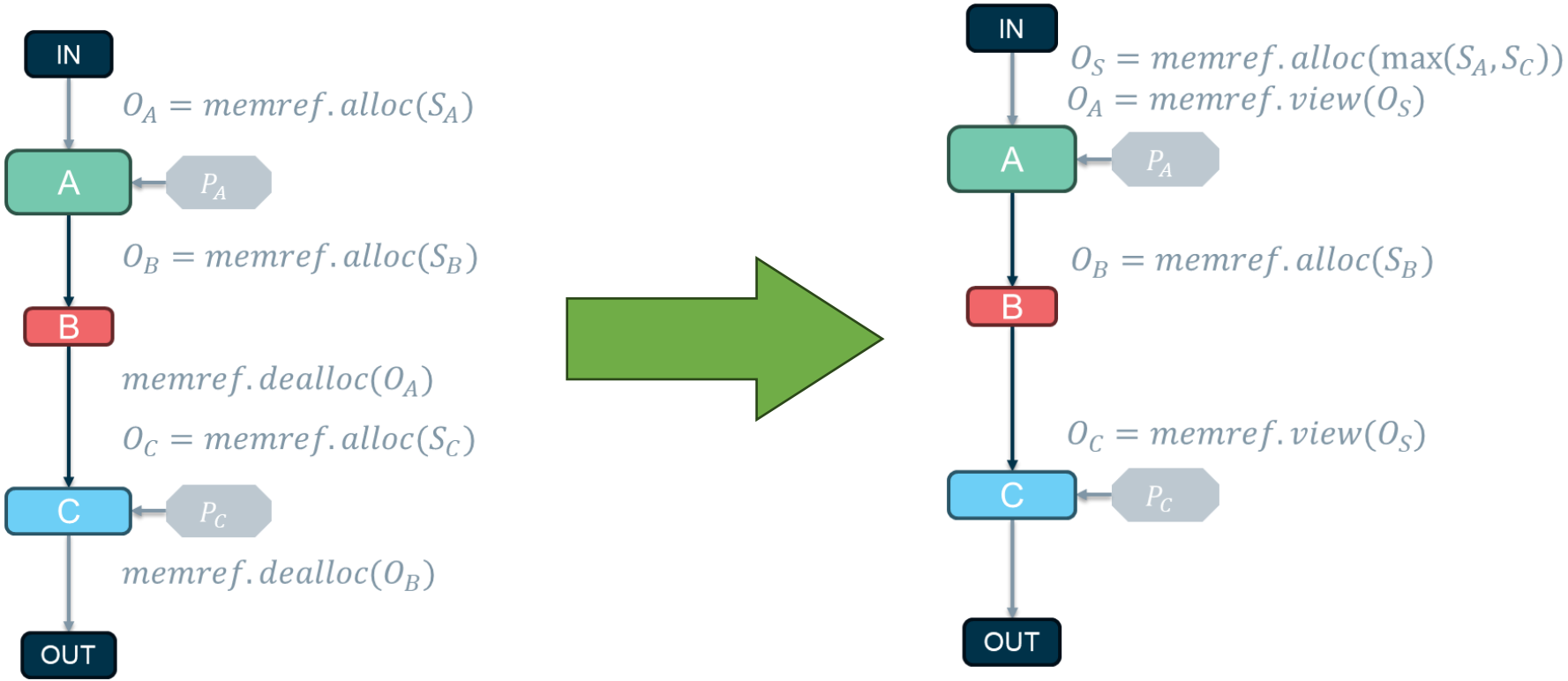


Memory Exclusion Graph (MEG)

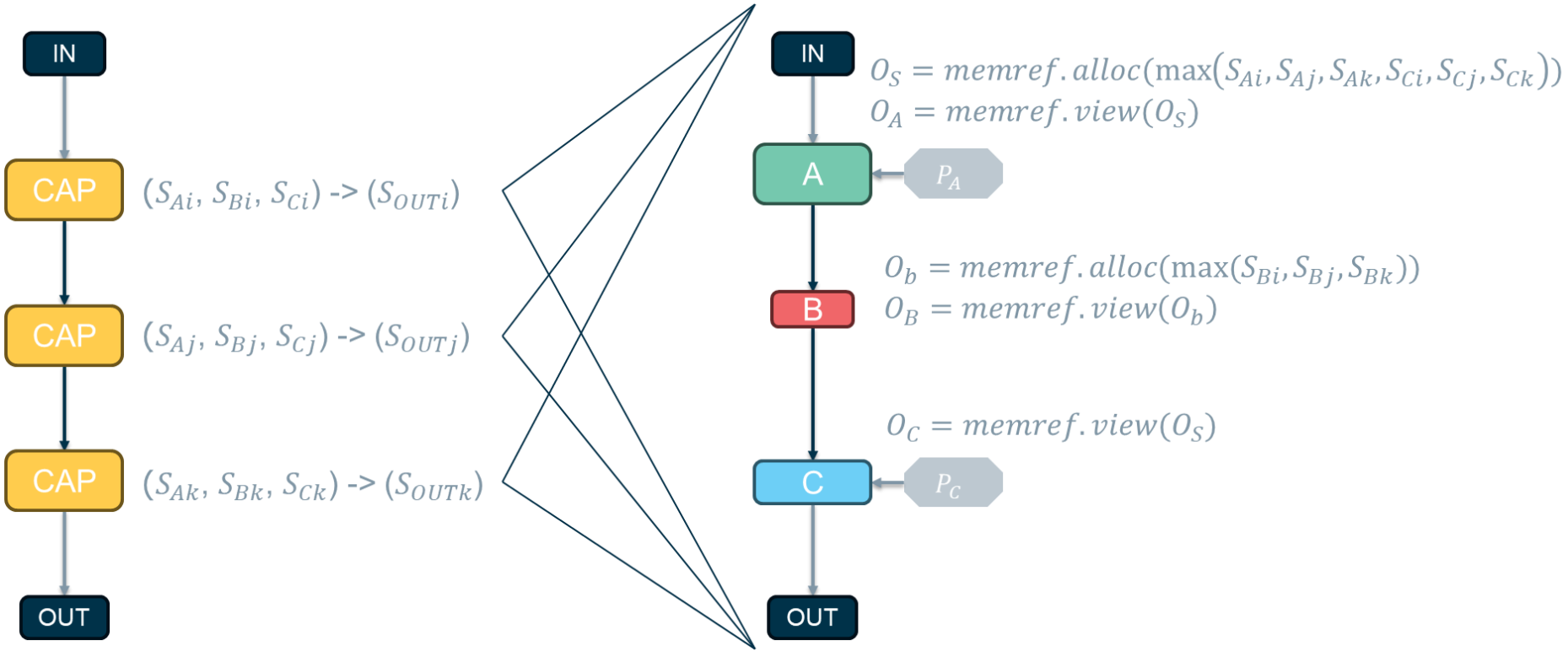


- **Input representation is a Synchronous Dataflow (SDF) Graph with memory buffers**
- **A Memory Exclusion Graph (MEG) is derived from the SDF**
- **MEG represents memory each memory buffer size and conflicts with other buffers**
- **Minimum memory requirement depends on the operation scheduling**
- **Post-scheduling MEG interval coloring solution yields optimal static memory allocation**

Optimized Memory Allocation

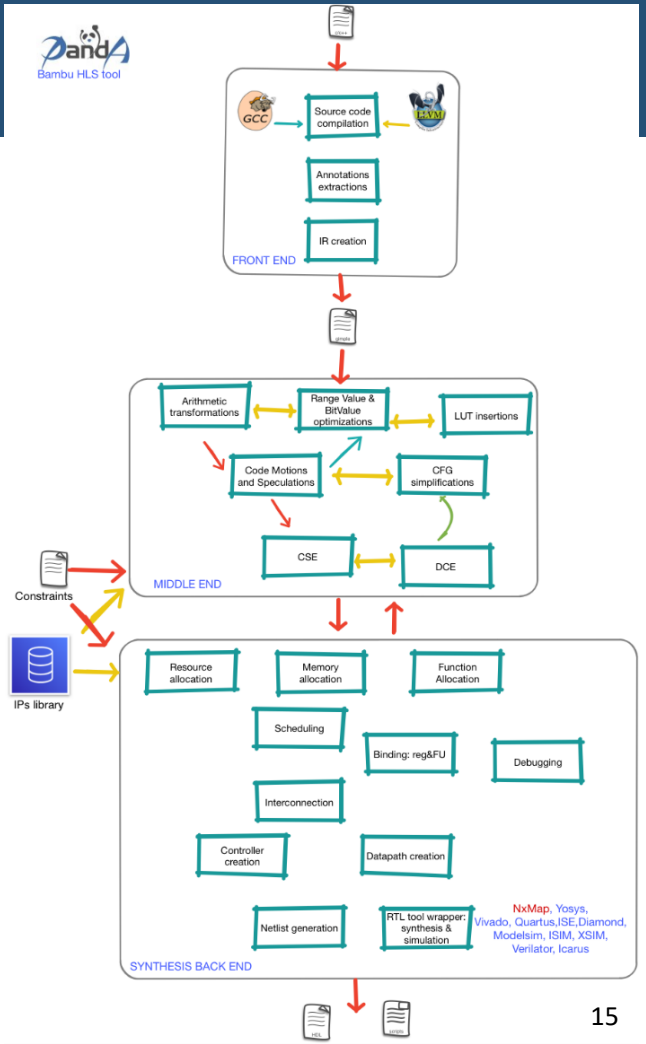


Static Memory Requirements



Bambu HLS

- HLS tools simplify the implementation of accelerators on FPGA
- HLS starts from high-level languages (C/C++)
  - Optimizes the Intermediate Representation
  - Allocates resources
  - Schedules operations
  - Binds them to the resources
  - Generates RTL descriptions for synthesis tools

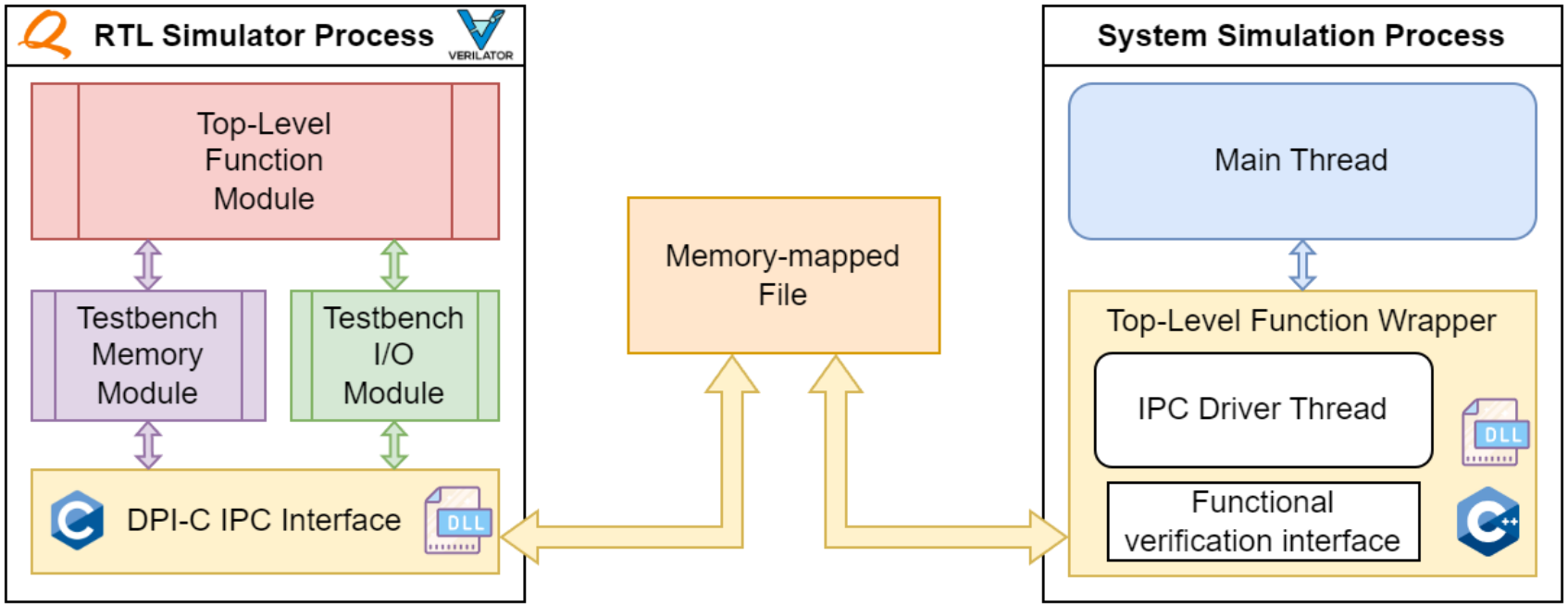


Bambu supporting AI/ML designs

- **Dataflow support extended: Point-to Set analysis**
- **Improved DSP support: multiplication fracturing**
- **Better support for memory inferencing: BRAM SDP/TDP/Async Mem**
- **Loop pipelining improved to better support AI/ML kernel**
- **Custom floating-point cores support (fp64, fp32, fp24, bfloat, fp16, fp8, …)**
- **Better support for interfaces: AXI4, AXI-Stream, FIFOs, DMA transfers, caches**
- **Advanced HW/SW Co-Simulation Environment**

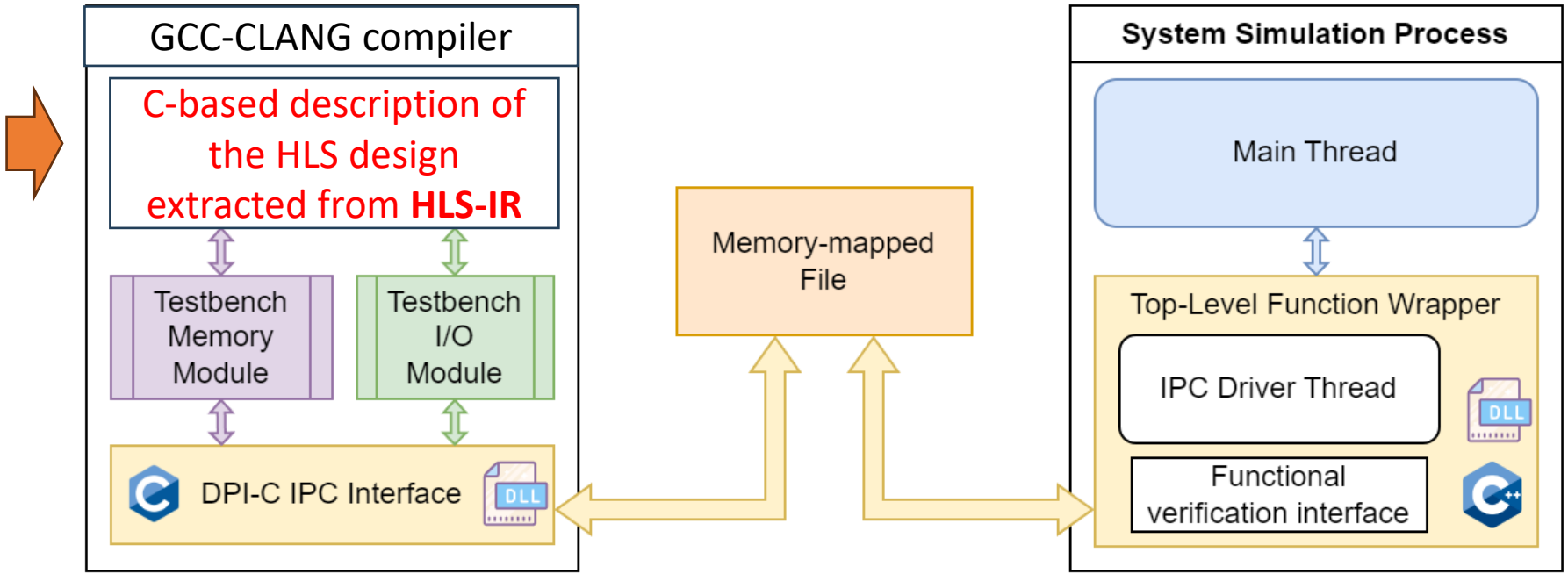
Speeding up co-simulation

- **Verilog/VHDL-C Co-simulation Environment**



10-20x Speedup raising kernel to C-based model

- **Cycle-accurate simulation through C annotated with timing information**



Questions

https://panda.dei.polimi.it
https://github.com/ferrandi/PandA-bambu

This work has been partially supported by the OSIP programme, Campaign “New concepts for onboard software development”, Idea Id I-2022-01122, Contract number 4000140887