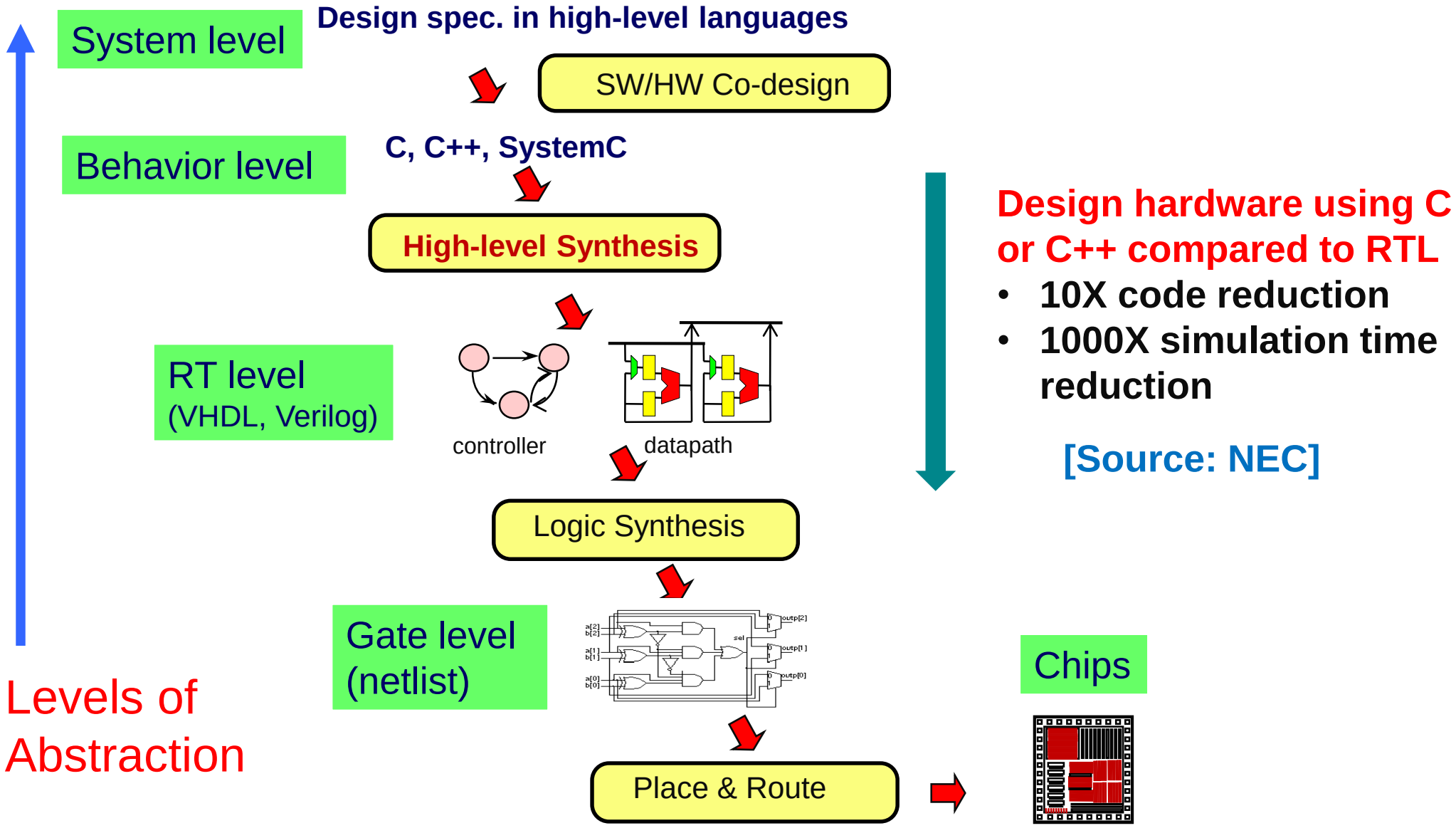
The Trend: High-Level Synthesis (HLS)



Current Challenges of High-Level Synthesis

- **HLS users often manually annotate their code with HLS pragmas or directives to give HLS compiler hints on parallelism and desirable synthesis approaches**
  - These pragmas have a significant impact on the performance and energy efficiency of the synthesis output.
- **Oftentimes manual code transformation and rewriting are also needed to improve the outcome**
  - Quality of HLS highly depends on how the code is written and how the HLS pragmas are added to the code.
  - A considerable amount of engineering time to iteratively adjust the source code and pragmas used.
  - Complicated applications or thorough optimizations may lead to long source code: difficult to read and maintain.
- **Applications are typically developed by domain experts with languages at a higher level of abstraction, e.g., Python for deep learning models developed by physicists or neuroscientists**
  - Such high-level programming models and styles focus more on describing the algorithm itself.
  - Needs a lowering step from high-level abstraction to low-level C/C++ code: time consuming and error-prone.
- **Existing intermediate representations (IR) of HLS code are originally designed for software compilation**
  - Cannot represent HLS intrinsic design hierarchy well.
  - Cannot perform HLS optimizations across different levels of abstractions.
- **Existing HLS leave many important HLS optimizations, such as task/module level resource-sharing and parallelization, hardware IP integration, and loop level analysis and transformation, to human designers being done manually**
  - Not productive and scalable enough to deal with large HLS designs.

- **Our solution: New HLS flows and frameworks -- PyLog and ScaleHLS**

PyLog: Algorithm-Centric FPGA Programming with Python

PyLog Source Code

```
@pylog
def accel(a, b, c, d, e):
  d = (a + b) * c
  e = map(lambda x, y: x+y), a, d)
```

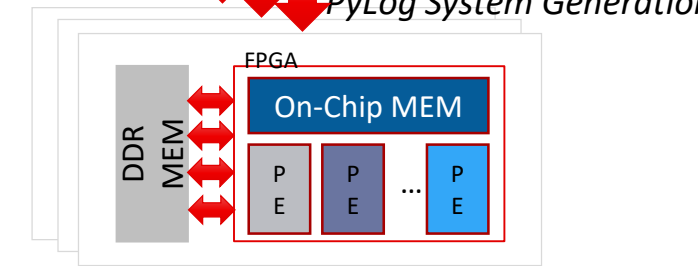
PyLog HLS C Code Generation

Optimized HLS C Code

```
void accel(int32* a, int32* b,
int32* c, int32* d, int32* e) {
  for(int i = 0; i < 32; i++) {
    for (ii...)
    #pragma HLS ...
  }
  for(int j = 0; j < 32; j++) {
    for (jj...)
  }
  ...
}
```

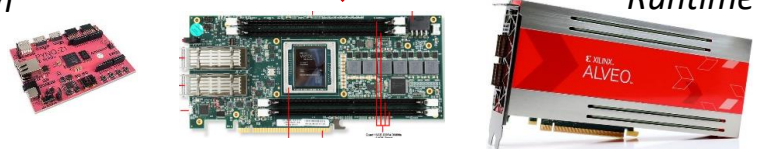
PyLog System Generation

System Design



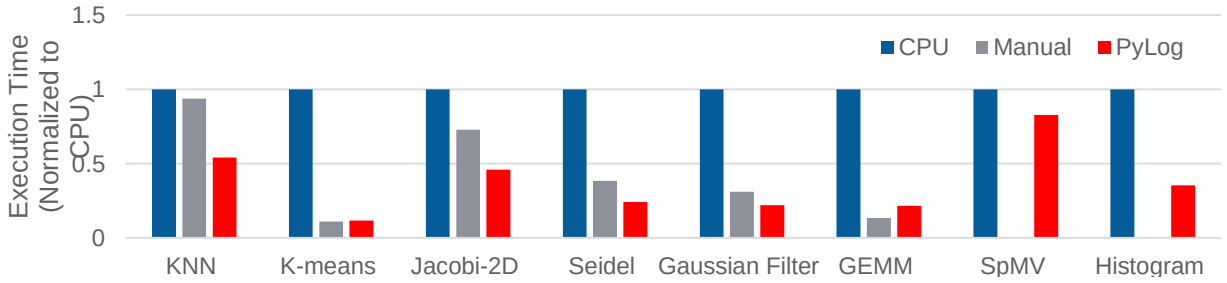
PyLog Runtime

Target Platform Deployment

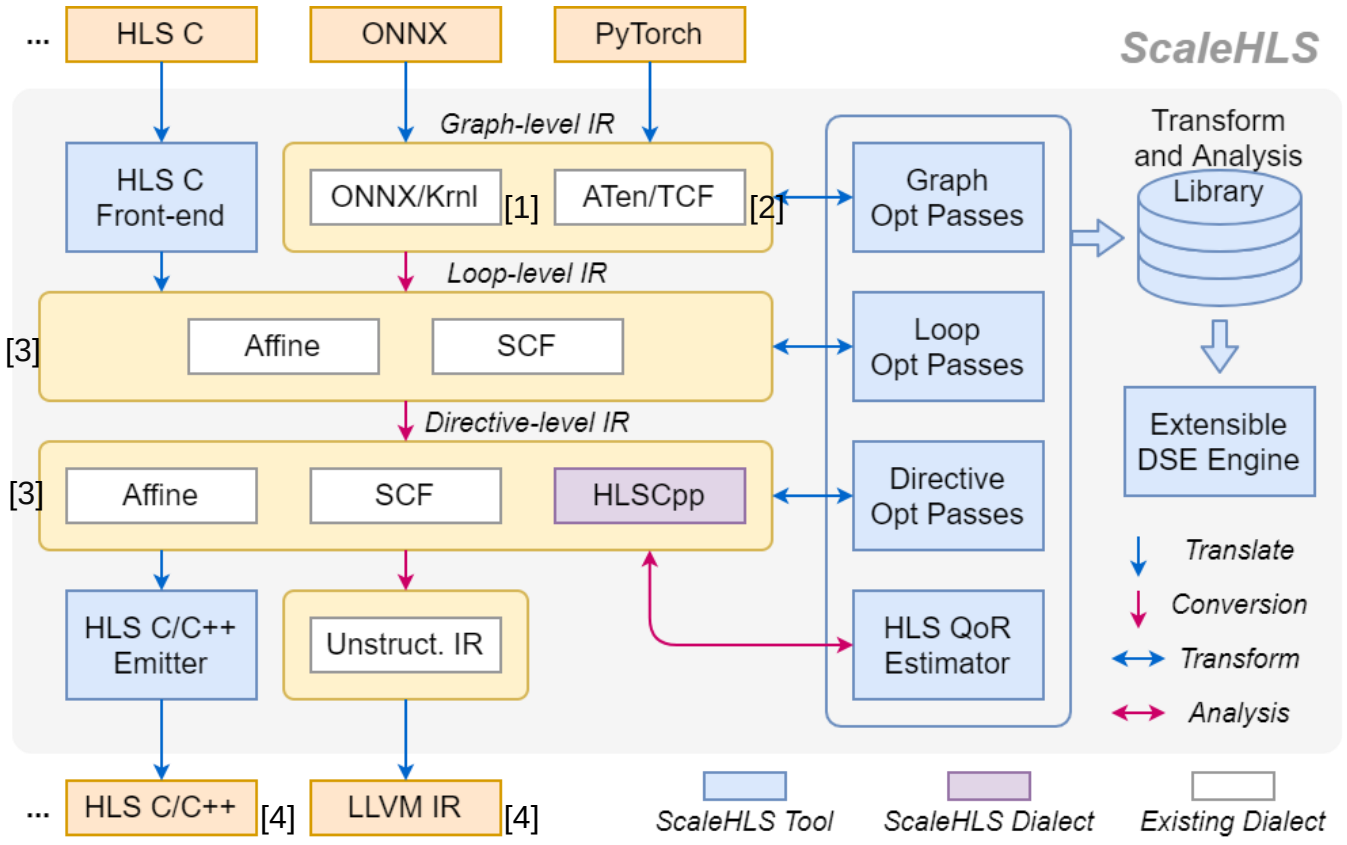


PyLog Flow (high level)

- PyLog provides a ***high-level*** abstraction of FPGA programming
- PyLog has built-in type inference engine and HLS optimizer
- On average ***3.17x*** and ***1.24x faster*** than optimized CPU and manually tuned FPGA accelerators
- PyLog can potentially be used in ***various scenarios***
  - Providing services at different levels (IPs, accelerator, application, compiler, etc.) in cloud environment
  - System designers create complete CPU+FPGA accelerators
  - Algorithm designers deploy algorithms/apps in HW prototypes
  - Enthusiast/students experimenting with FPGAs in DIY projects
  - and many more



ScaleHLS Framework



[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. https://github.com/onnx/onnx-mlir
[2] NPComp: MLIR based compiler toolkit for numerical python programs. https://github.com/llvm/mlir-npcomp
[3] MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir
[4] Vitis HLS LLVM Front-end: https://github.com/Xilinx/HLS

Multi-level Representation

Graph-level IR: ONNX [1] and ATen [2] dialect.

Loop-level IR: MLIR built-in Affine dialect [3].

Directive-level IR: A customized HLSCpp dialect to represent HLS-specific structures and directives.

Multi-level Optimization

Optimization Passes: Cover the graph, loop, and directive levels. Solve HLS optimization problems at the 'correct' abstraction level.

QoR Estimator: Estimate the latency and resource utilization through IR analysis.

Automated Design Space Exploration

Transform and Analysis Library: Parameterized interfaces of all optimization passes and the QoR estimator. A great playground of DSE.

Automated DSE Engine: Find the Pareto-frontier of the throughput-area trade-off design space.

End-to-End Compilation Flow

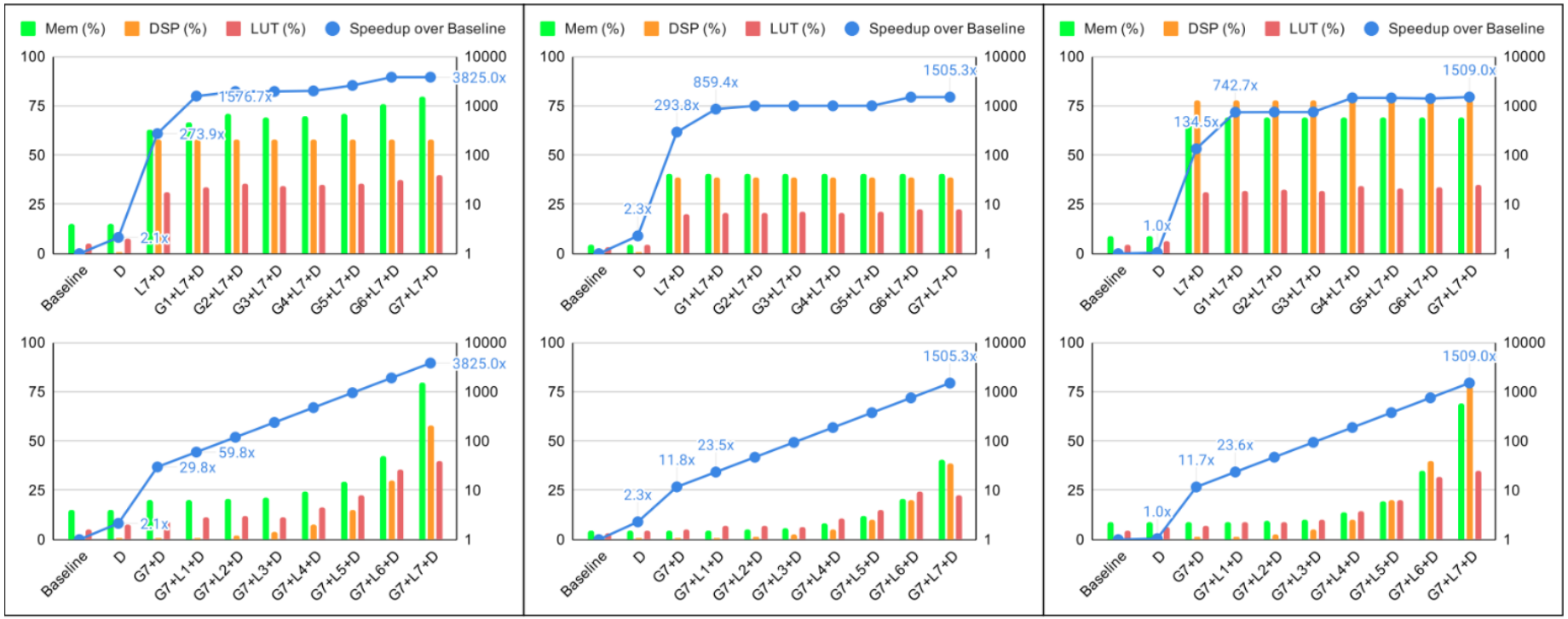
HLS C Front-end: Parse C programs into MLIR.

HLS C/C++ Emission Back-end: Generate optimized HLS code for downstream tools, such as Vitis HLS [4].

ScaleHLS Results on PyTorch DNN Models

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	FF (SLR Util. %)	Our DSP Effi. (OP/Cycle/DSP)	DSP Effi. of TVM-VTA [32]
ResNet-18	3825.0×	60.8	91.7Mb (79.5%)	1326 (58.2%)	157902 (40.1%)	54766 (6.9%)	1.343	0.344
VGG-16	1505.3×	37.3	46.7Mb (40.5%)	878 (38.5%)	88108 (22.4%)	31358 (4.0%)	0.744	0.296
MobileNet	1509.0×	38.1	79.4Mb (68.9%)	1774 (77.8%)	138060 (35.0%)	56680 (7.2%)	0.791	0.468

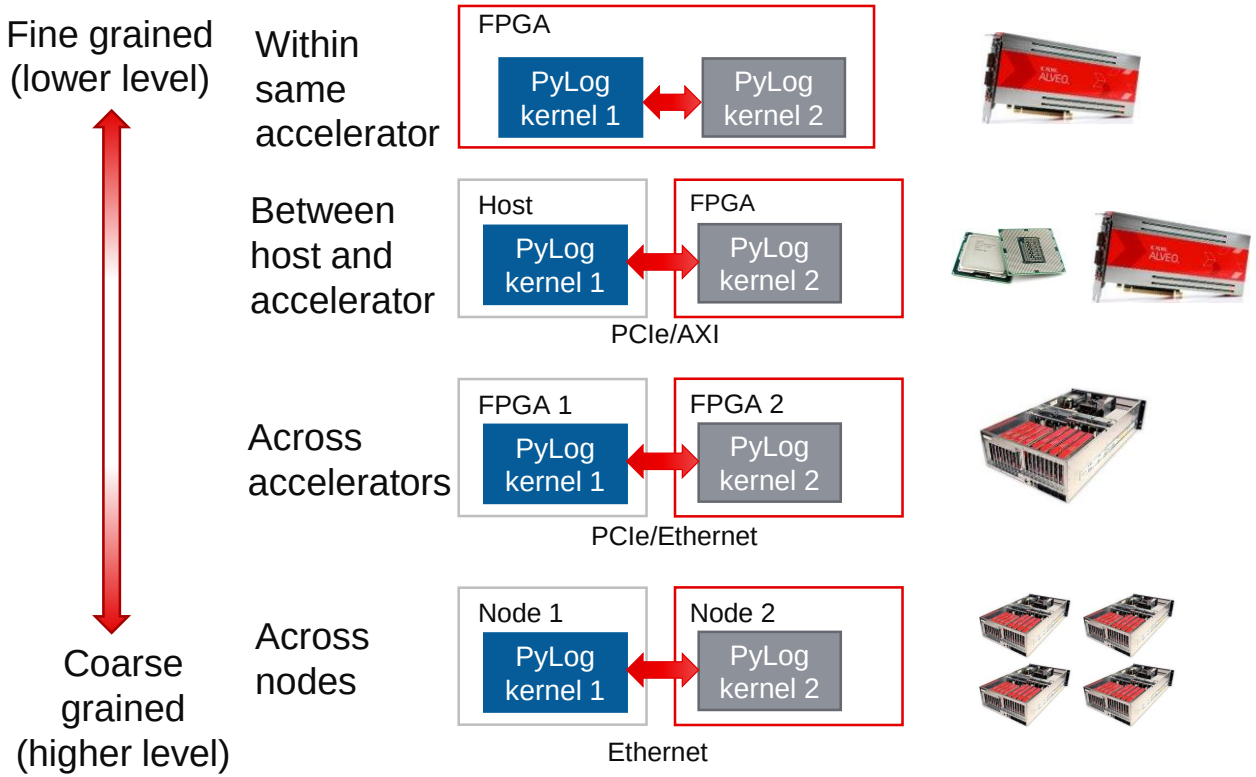
Evaluation results on Xilinx VU9P FPGA with graph, loop, and directive optimizations. *Speedup* is with respect to the baseline designs compiled from PyTorch to HLS C++ by ScaleHLS but without the multi-level optimization.



Ablation study of DNN models. D, $L\{n\}$, and $G\{n\}$ denote directive, loop, and graph optimizations, respectively. Larger n indicates larger loop unrolling factor and finer dataflow granularity for loop and graph optimizations, respectively.

Future for PyLog

- Support various heterogeneous system organizations, in *cloud computing* setting, targeting A3D3 workloads
- Program, optimize hardware interfaces at Python level
- Extend the abstraction to other accelerators, e.g., GPU



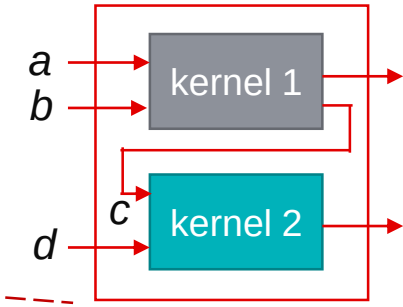
Programming heterogeneous system organizations

Interfaces and interconnects at Python level

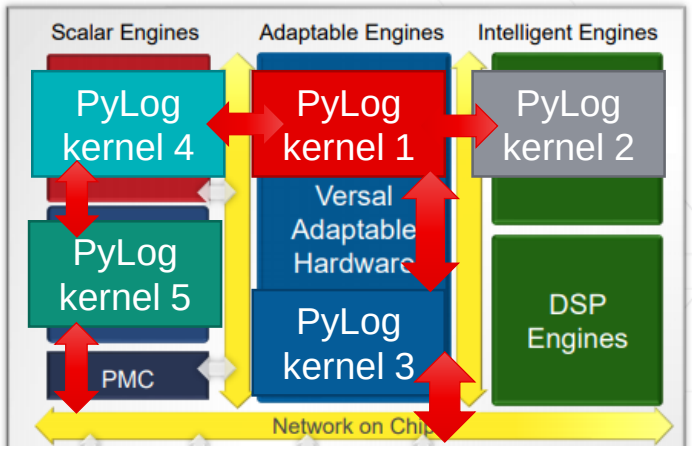
```
@pylog
def acc1(a, b):
    ...
@pylog
def acc2(c, d):
    ...
if __name__ == "__main__":
    a = np.array([1, 3, ...])
    b = np.array([8, 9, ...])
    d = np.array([8, 9, ...])
    combine(acc1, acc2)(a, b, d)
```

kernel 1 *kernel 2* *host*

- Returns a new function that combines kernels
- Connects inputs and outputs with kernels



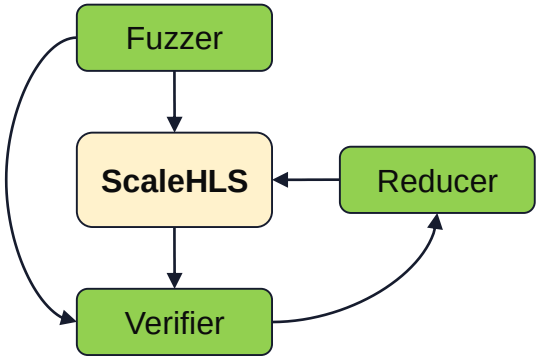
Latest heterogeneous platforms

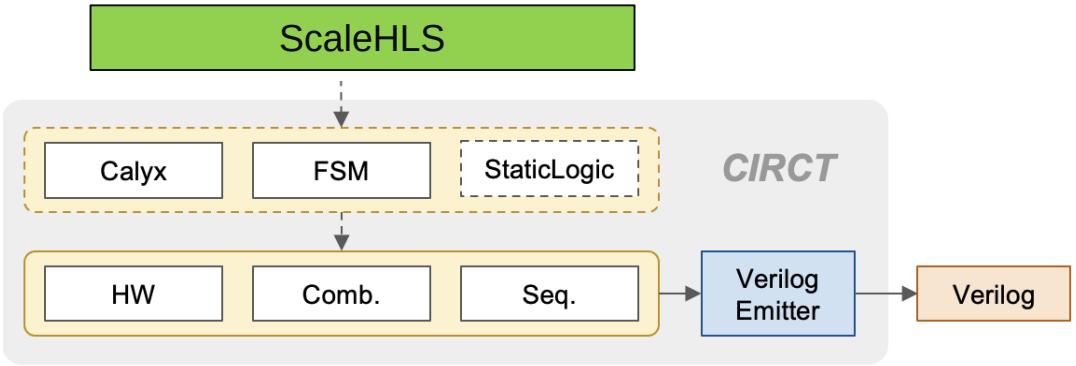


Xilinx Versal devices

Future for ScaleHLS

- Verify the correctness of ScaleHLS conversions, optimizations, and code generation.
- Reduce the targeted program to locate the bugs.





- A direct RTL code generation engine to exploit RTL-level optimizations to further improve the QoR of the designs.
- The directive-level IR in ScaleHLS can be converted to hardware description IRs in CIRCT and finally be emitted as SystemVerilog designs.

- Generate new hardware IPs automatically through the ScaleHLS compilation flow.
- Represent, integrate, and parameterize hardware IPs within MLIR.

- Predict the performance and resource utilization using machine learning (ML) model and train the model with the evaluated design points.
- Inference the ML model and use ScaleHLS transform library to generate the proposed design.