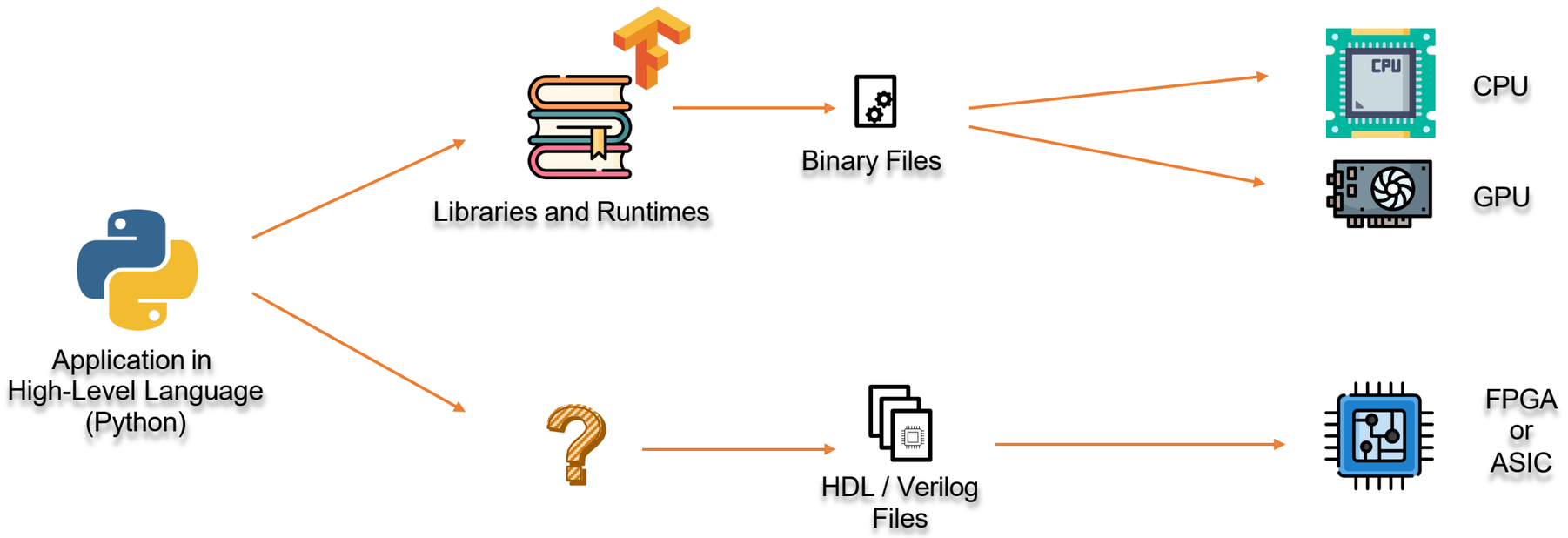
MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration

Based on ICCAD ‘22 paper and ISCA ’24 Tutorial
https://dl.acm.org/doi/10.1145/3508352.3549424
https://hpc.pnl.gov/SODA/tutorials/2024/ISCA24.html

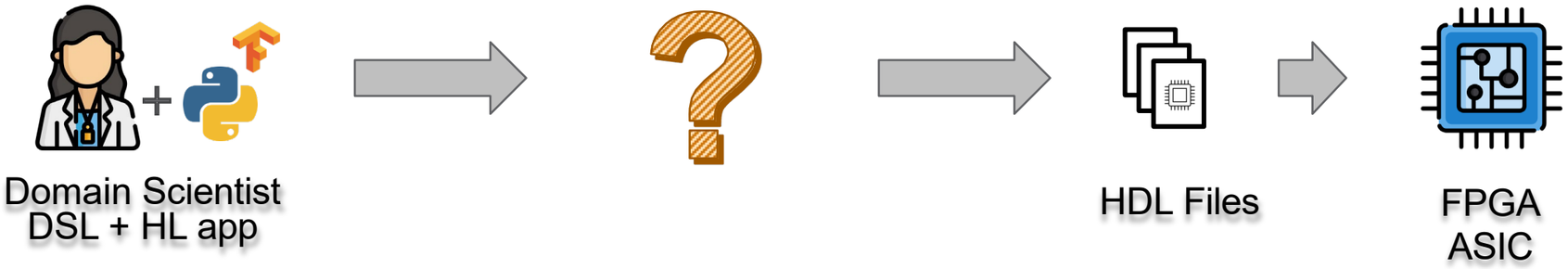
Qucheng Jiang

Agile Hardware Design and Prototyping

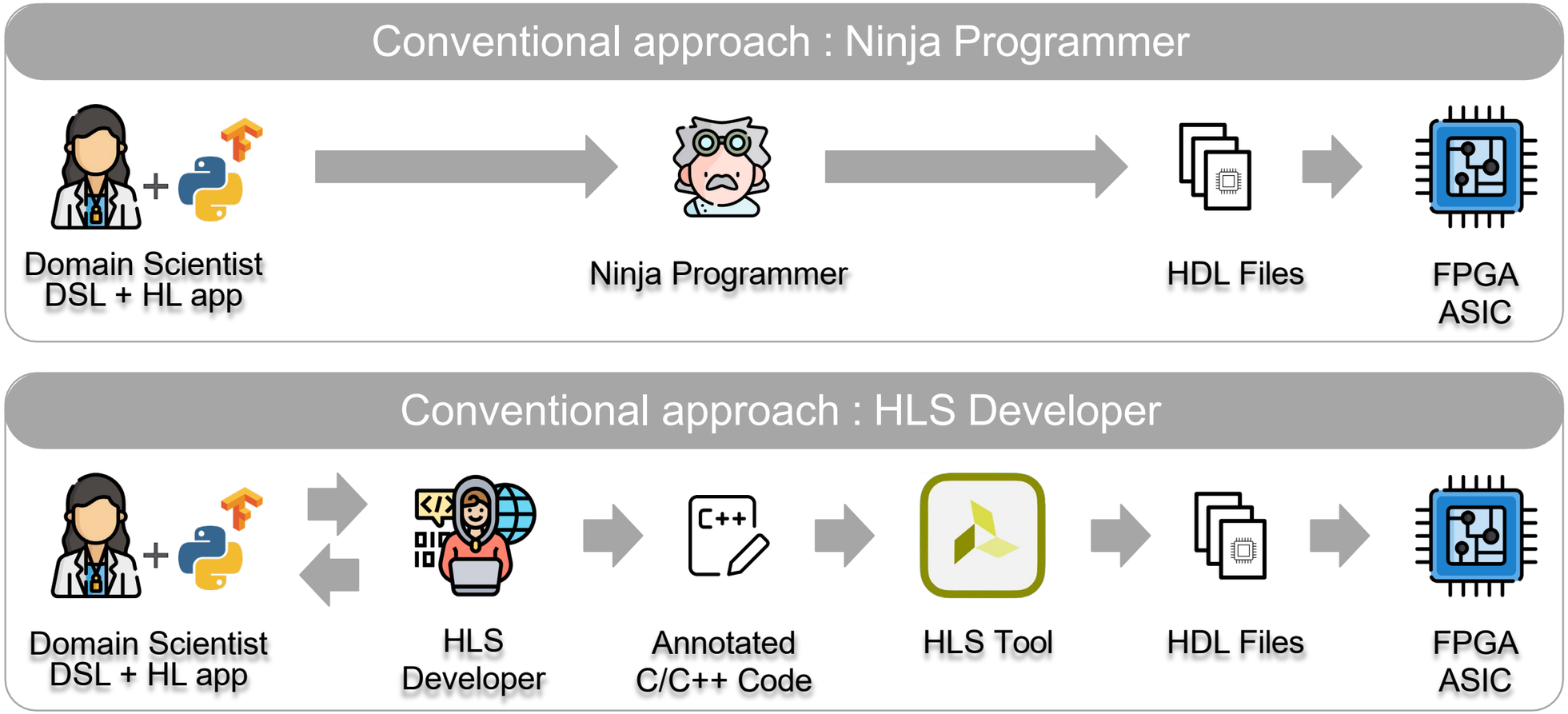


Current challenge

- It is challenging **to map** applications into custom hardware
- It is challenging **to extract performance** of the custom hardware
- Can we transform the Domain Scientist in a "**Lead User**" for Custom Domain Specific Accelerators?



Current workflow



Related Work

Tool	Input	Backend tools	F1	F2	F3	F4	F5
ScaleHLS	MLIR (Affine or Higher)	Vivado HLS	✓	✗	✓	✓	✗
CIRCT HLS	MLIR (Affine or Higher)	CIRCT	✓	✗	✗	✗	✓
FROST	Halide, Tiramisu	Vivado HLS	✗	✗	✗	✓	✗
Hot & Spicy	Annotated Python	SDSoC	✗	✗	✗	✓	✗
HeteroCL	Annotated C, OpenCL	Intel or Vitis HLS	✗	✗	✗	✓	✗
HPVM2FPGA	HeteroC++	Intel HLS	✓	✓	✓	✓	✗
Phism	C/C++	Vitis HLS	✓	✗	✗	✓	✗

F1 Optimizations Without Manual Annotations

F2 Automatic Partitioning of Host and Kernel Code

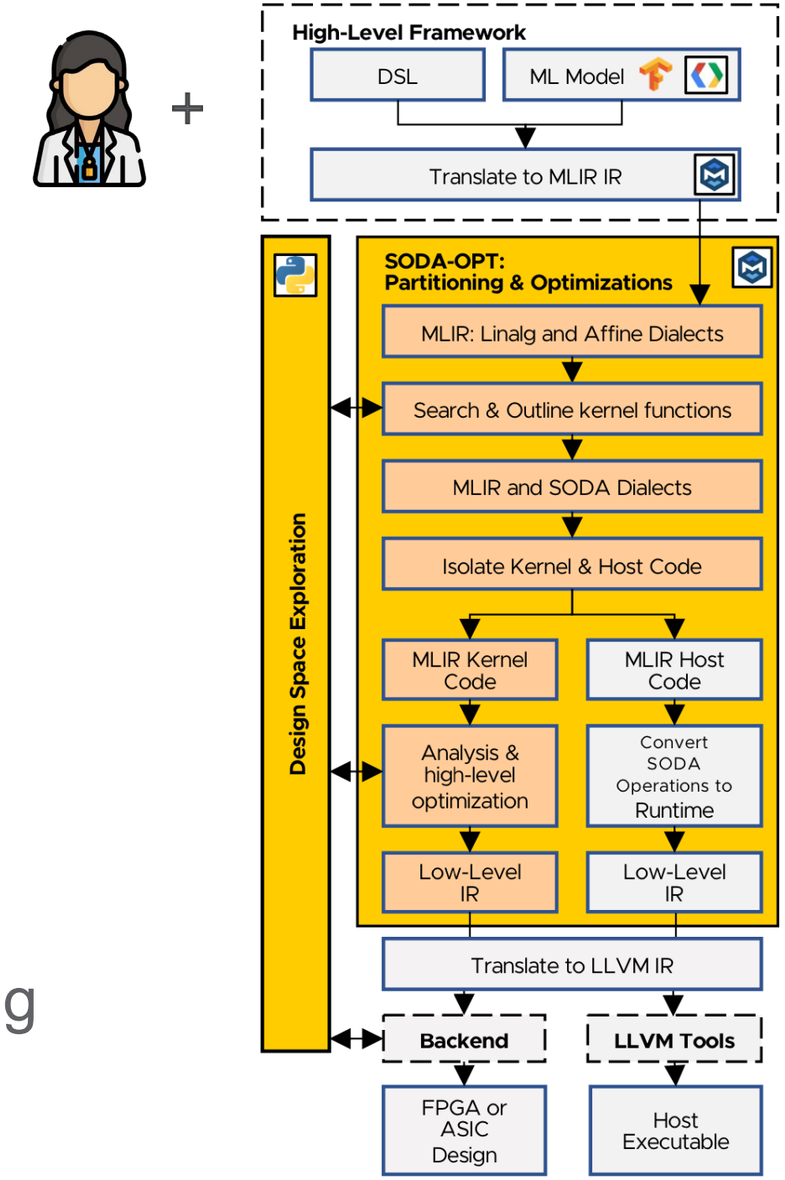
F3 Design Space Exploration of Optimization Strategies

F4 FPGA Support

F5 ASIC Support

SODA-OPT contributions / scope

- **MLIR-**based **inputs**
  - Supports all high-level application once it can be converted into `linalg`, `affine` dialects
- Configurable pipeline of **high-level optimization** passes
- **Automated flow** from high-level outlined kernels into custom hardware impl. through HLS backends
- **DSE** of compiler options: pipeline param. tuning



The Multi-Level Intermediate Representation Compiler Infrastructure

- **Open-source**
- **Progressive lowering** between existing and new operations
- **Reuse** of abstractions and compiler transformations
- Enables **co-existence** of different abstractions

Figure from: Lattner, Chris, et al. "MLIR: Scaling compiler infrastructure for domain specific computation." *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021.

MLIR Dot Product Example and Lowering

Linalg abstraction

```
func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
               %out: memref<f32>) {
  linalg.dot ins(%A, %B: memref<100xf32>, memref<100xf32>)
             outs(%out: memref<f32>)
  return
}
```

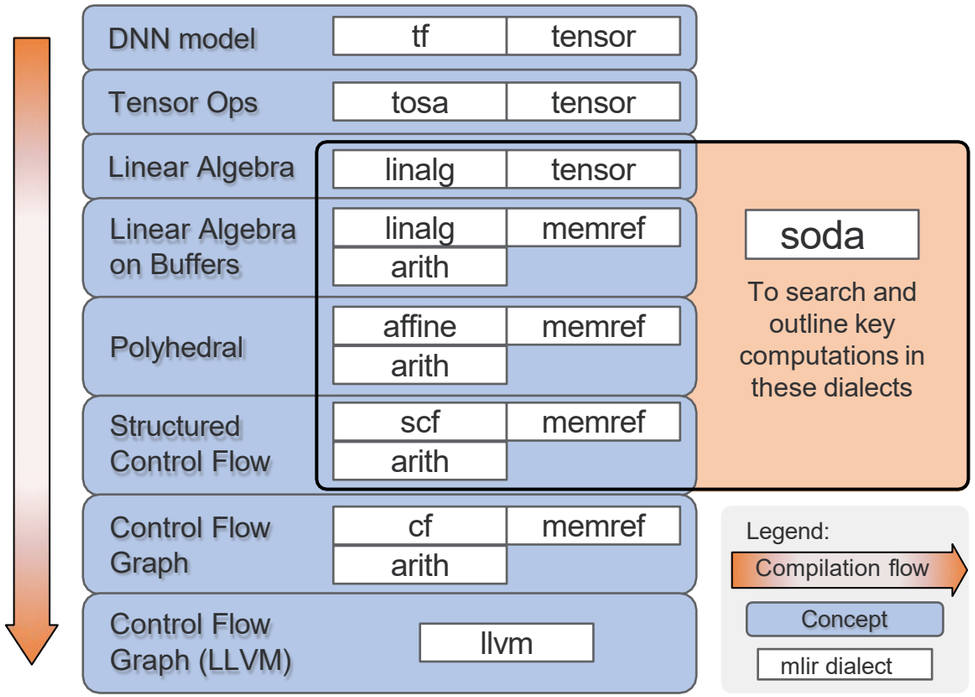
SCF abstraction

```
func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
               %out: memref<f32>) {
  %c0 = arith.constant 0 : index
  %c100 = arith.constant 100 : index
  %c1 = arith.constant 1 : index
  scf.for %arg3 = %c0 to %c100 step %c1 {
    %0 = memref.load %A[%arg3] : memref<100xf32>
    %1 = memref.load %B[%arg3] : memref<100xf32>
    %2 = memref.load %out[] : memref<f32>
    %3 = arith.mulf %0, %1 : f32
    %4 = arith.addf %2, %3 : f32
    memref.store %4, %out[] : memref<f32>
  }
  return
}
```

CF abstraction

```
func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
               %out: memref<f32>) {
  %c0 = arith.constant 0 : index
  %c100 = arith.constant 100 : index
  %c1 = arith.constant 1 : index
  cf.br ^bb1(%c0 : index)
^bb1(%0: index):  // 2 preds: ^bb0, ^bb2
  %1 = arith.cmpi slt, %0, %c100 : index
  cf.cond_br %1, ^bb2, ^bb3
^bb2:  // pred: ^bb1
  %2 = memref.load %A[%0] : memref<100xf32>
  %3 = memref.load %B[%0] : memref<100xf32>
  %4 = memref.load %out[] : memref<f32>
  %5 = arith.mulf %2, %3 : f32
  %6 = arith.addf %4, %5 : f32
  memref.store %6, %out[] : memref<f32>
  %7 = arith.addi %0, %c1 : index
  cf.br ^bb1(%7 : index)
^bb3:  // pred: ^bb1
  return
}
```

The SODA Dialect



Before outlining (used in the search step)

`soda.launch`	Marks code blocks to be outlined and extracted into separate kernel functions
`soda.terminator`	Indicates the end of a code region to be outlined

After outlining

`soda.launch_func`	Calls outlined functions and is replaced by accelerator call API in the host code
`soda.module`	Holds the list of outlined kernels to be later optimized
`soda.func`	Defines outlined kernels and their interface as functions
`soda.return`	Indicates the end of a kernel function

MLIR Operations | Semantics

Search and Outlining

Before outlining (used in the search step)

`soda.launch`	Marks code blocks to be outlined and extracted into separate kernel functions
`soda.terminator`	Indicates the end of a code region to be outlined

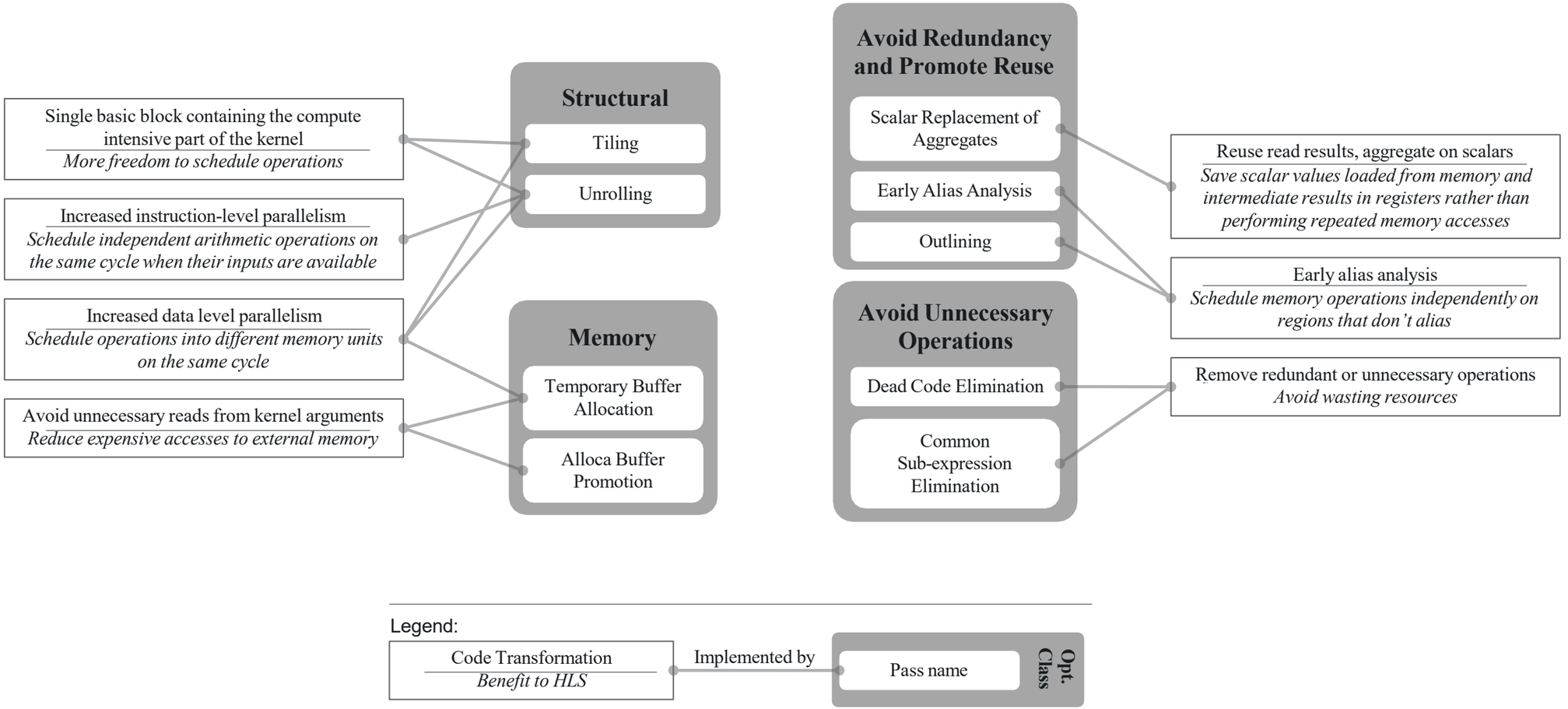
After outlining

`soda.launch_func`	Calls outlined functions and is replaced by accelerator call API in the host code
`soda.module`	Holds the list of outlined kernels to be later optimized
`soda.func`	Defines outlined kernels and their interface as functions
`soda.return`	Indicates the end of a kernel function

```
module {
  func.func @main(%A: memref<42x42xf32>, %B: memref<42x42xf32>,
                  %C: memref<42x42xf32>) {
    soda.launch {
      linalg.matmul ins(%A, %B : memref<42x42xf32>, memref<42x42xf32>)
                    outs(%C : memref<42x42xf32>)
      soda.terminator
    }
    return
}}
```

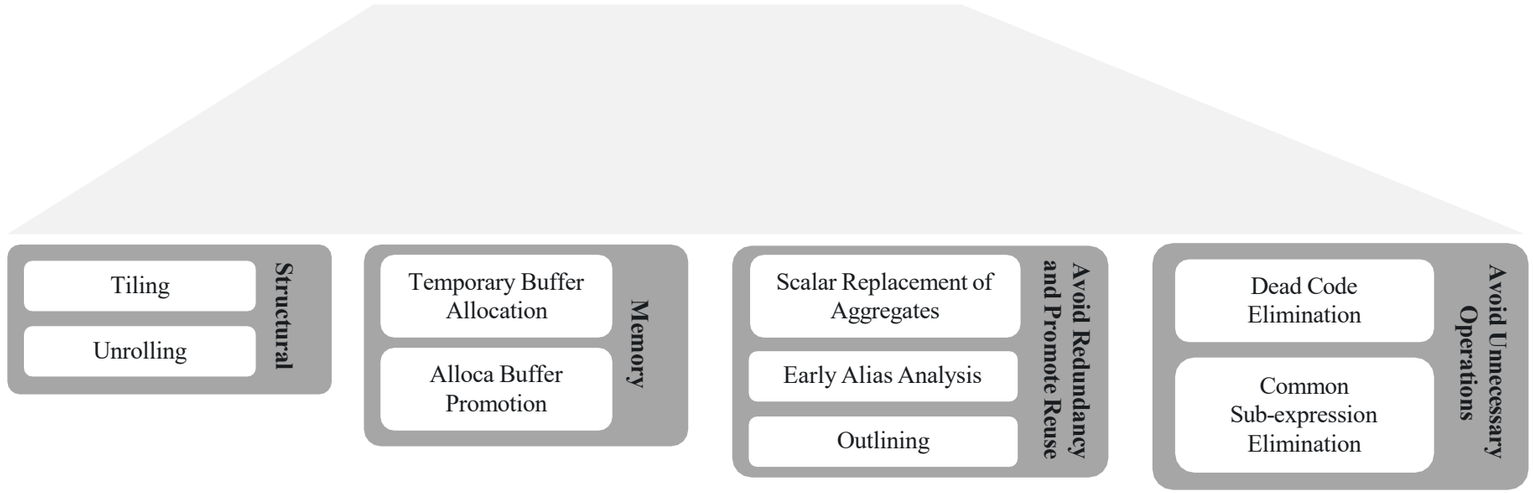
```
module attributes {soda.container_module} {
  func.func @my_matmul(%A: memref<42x42xf32>, %B: memref<42x42xf32>,
                       %C: memref<42x42xf32>) {
    soda.launch_func @accelerators::@my_matmul_kernel args(
      %A : memref<42x42xf32>, %B : memref<42x42xf32>, %C : memref<42x42xf32>)
    return
  }

  soda.module @accelerators {
    soda.func @my_matmul_kernel(%A_kernel: memref<42x42xf32>,
                                %B_kernel: memref<42x42xf32>,
                                %C_kernel: memref<42x42xf32>) kernel{
      linalg.matmul ins(%A_kernel, %B_kernel : memref<42x42xf32>, memref<42x42xf32>)
                    outs(%C_kernel : memref<42x42xf32>)
      soda.return
}}}
```

Optimizations for High-Level Synthesis



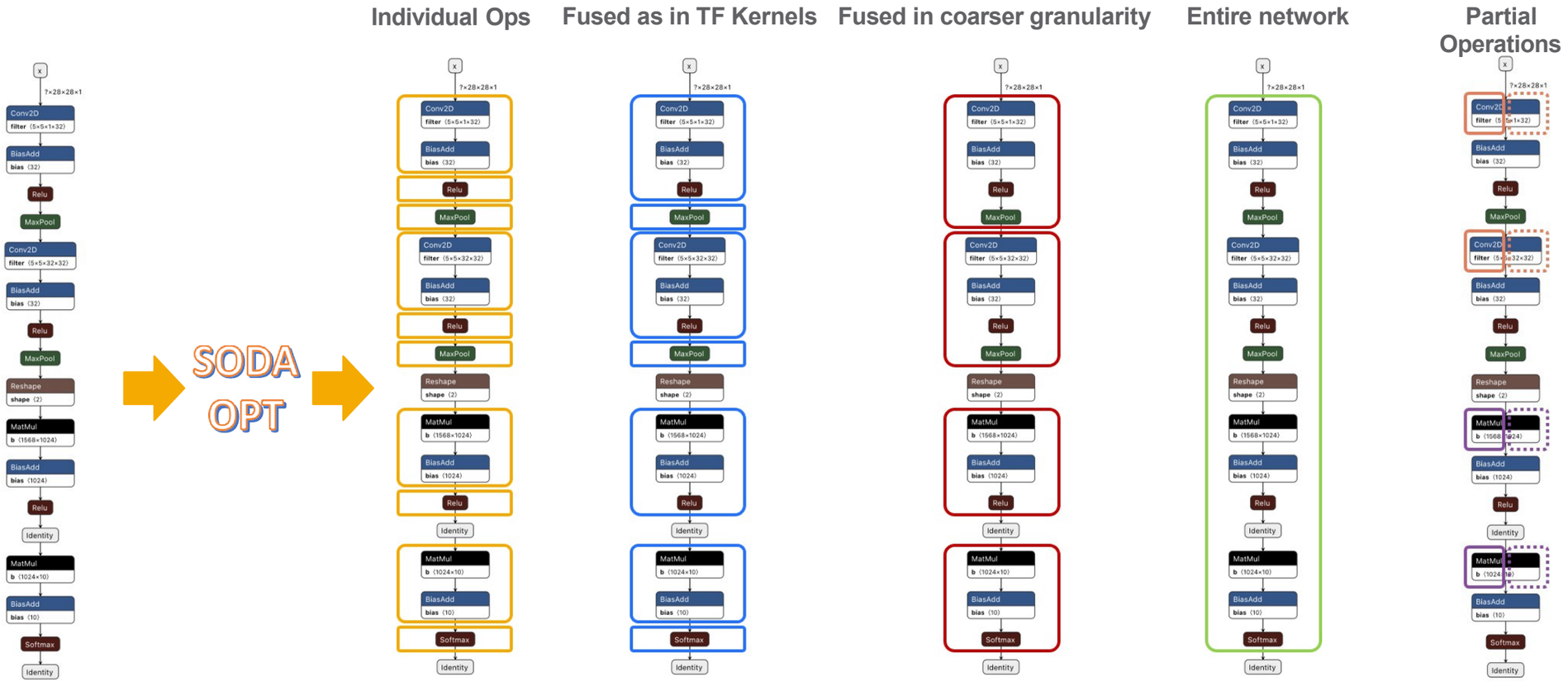
DSE Engine

DSE (Backend, Kernels, Metric 🎯, Transformations) = SODA-OPT pass pipeline for Target Backend



Outlining in Different Granularities

- The impact of outlining and generating accelerators for different granularities of a DNN model



Outlining in Different Granularities

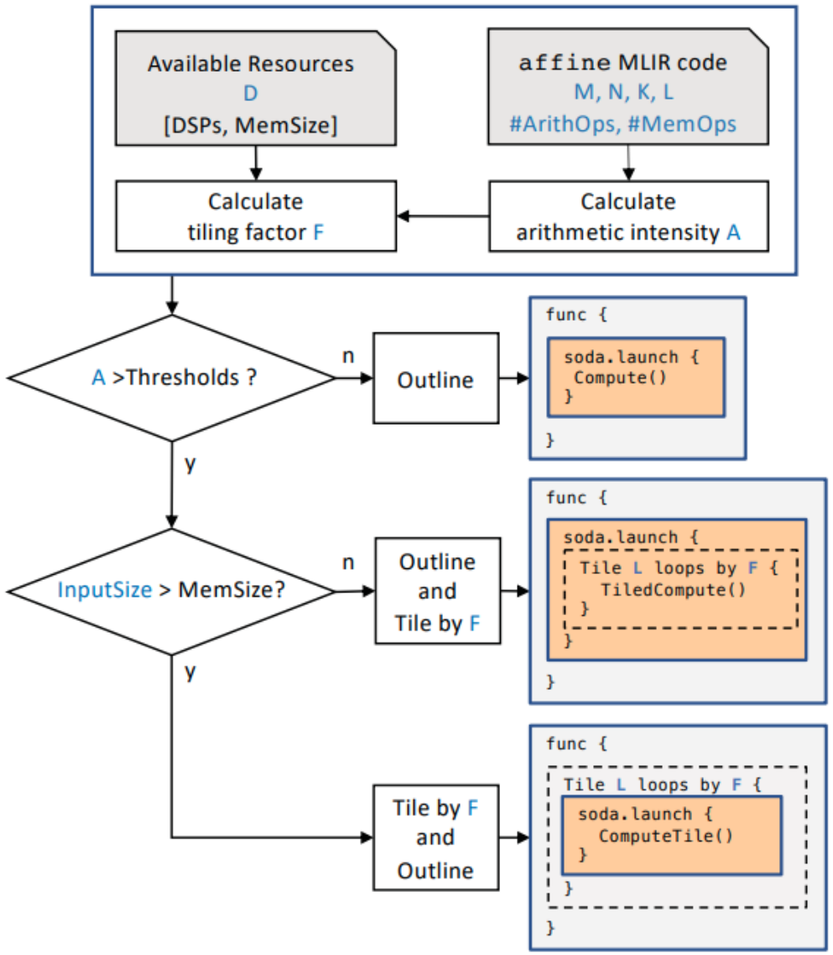
Execution times of LeNet with different outlining strategies.

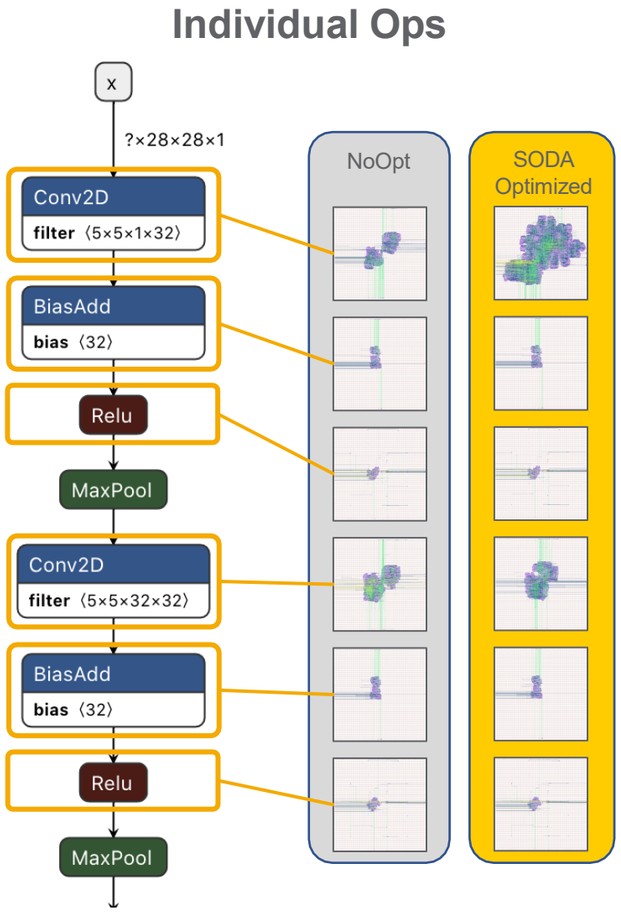
Layer type	Layer params	Individual Ops		Fused as in TF kernels		Fused in coarser granularity		Entire network	
		Baseline	Unrolled	Baseline	Unrolled	Baseline	Unrolled	Baseline	Unrolled
Conv2D	5x5,6,same	2,423,374	2,353,598	2,462,602	2,388,122	2,526,225	2,443,073	6,806,682	5,965,844
Activation	Relu	39,230	34,526						
AveragePooling2D	2x2, s2x2	88,244	84,338	88,244	84,338				
Conv2D	5x5,16,same	4,917,022	4,835,522	4,917,022	4,835,522	5,175,970	4,853,938		
Activation	Relu	12,912	11,312						
AveragePooling2D	2x2, s2x2	30,092	28,842	30,092	28,842				
Dense	120 units	1,010,522	926,402	1,011,602	927,362	1,011,602	927,362		
Activation	Relu	962	842						
Dense	84 units	213,446	195,722	214,202	196,394	214,202	196,394		
Activation	Relu	674	590						
Dense	10 units	17,841	16,372	17,841	16,372	19,084	16,731		
Activation	Softmax	454	410	454	410				
Total		8,754,773	8,488,476	8,742,059	8,477,362	8,947,083	8,437,498	6,806,682	5,965,844

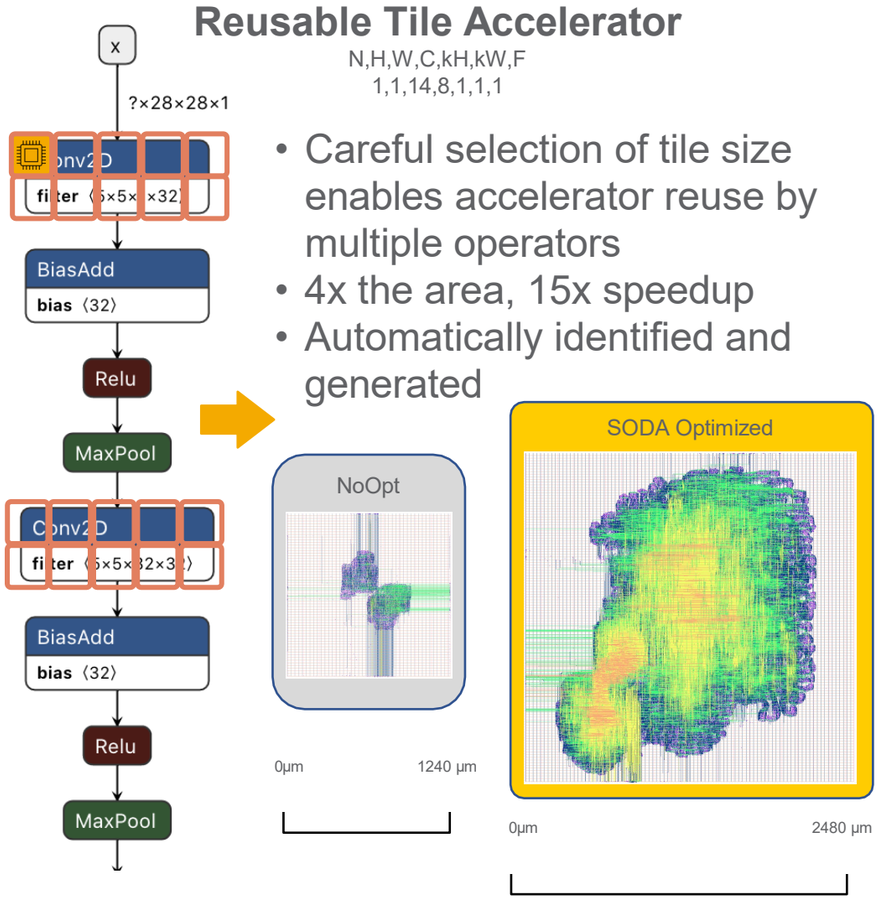
1.47x Speedup

* FPGA Target: Xilinx xc7vx690t-ffg1930-3 @ 100MHz
Merged cells correspond to a single accelerator.

Tiling, Outlining and Optimizing

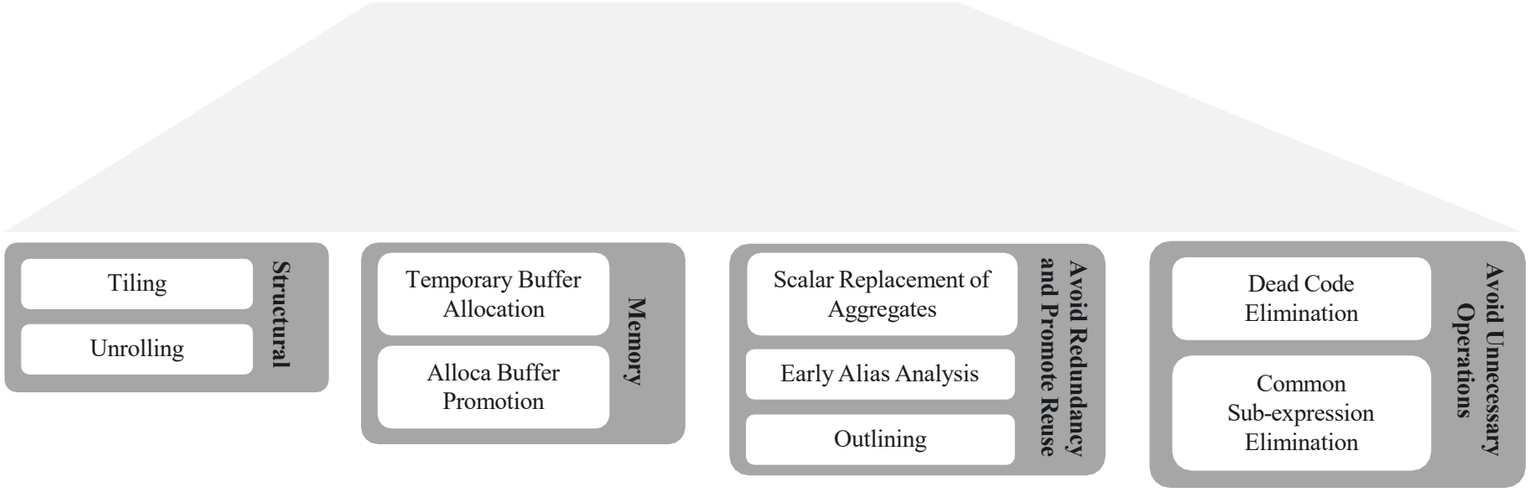




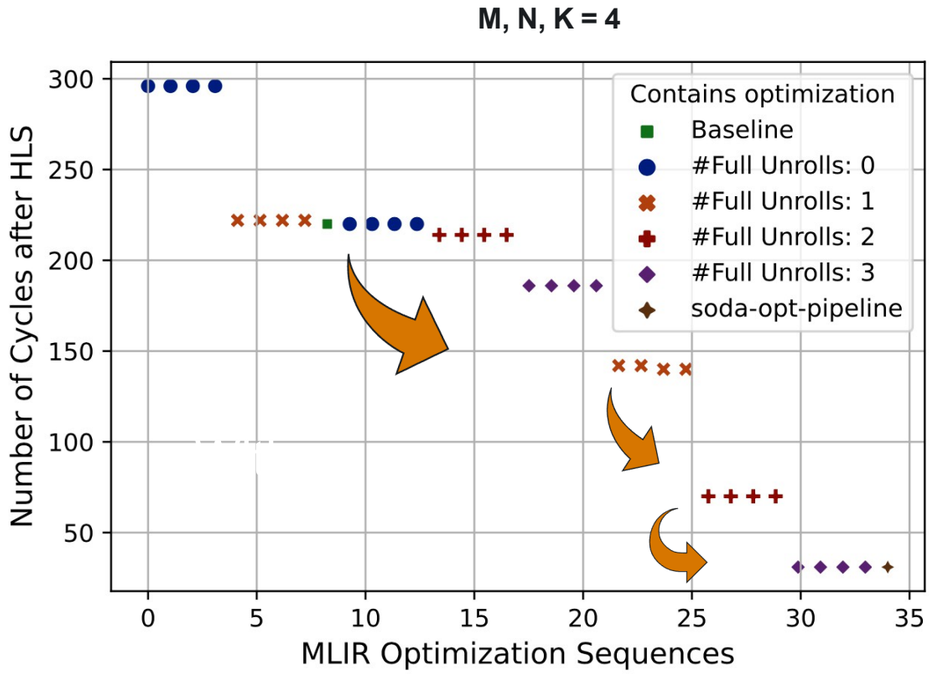


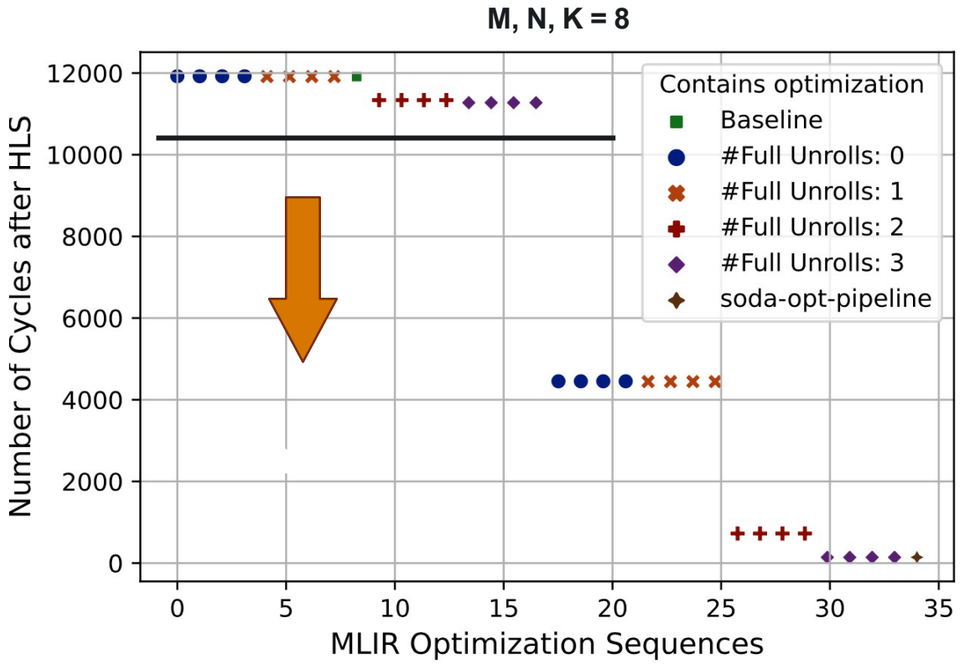
Effects of DSE Engine

DSE (Bambu, Kernels, Runtime 🎯 , Transformations) = SODA-OPT pass pipeline for Bambu



Effects of DSE Engine on GEMM Kernels

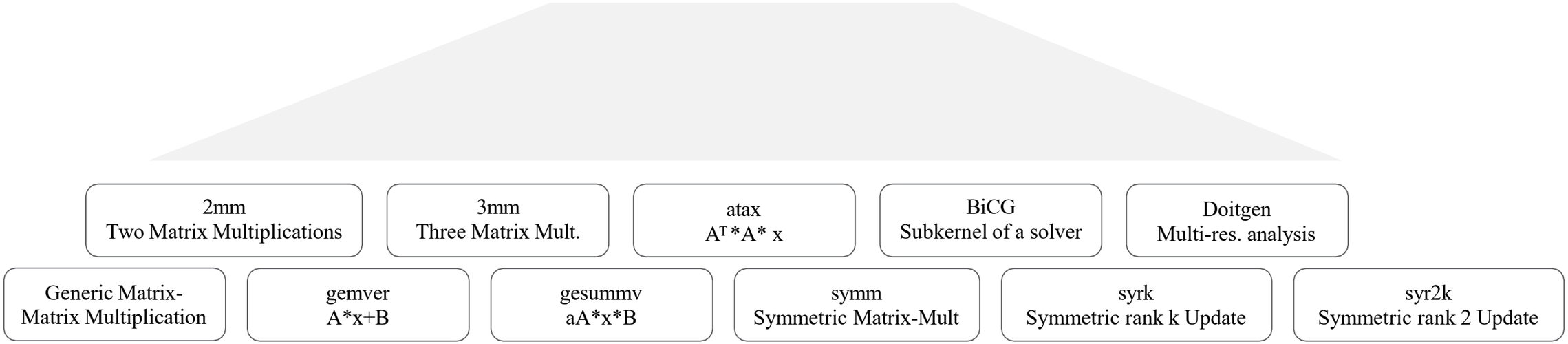




Structural Optimizations and Memory Optimizations

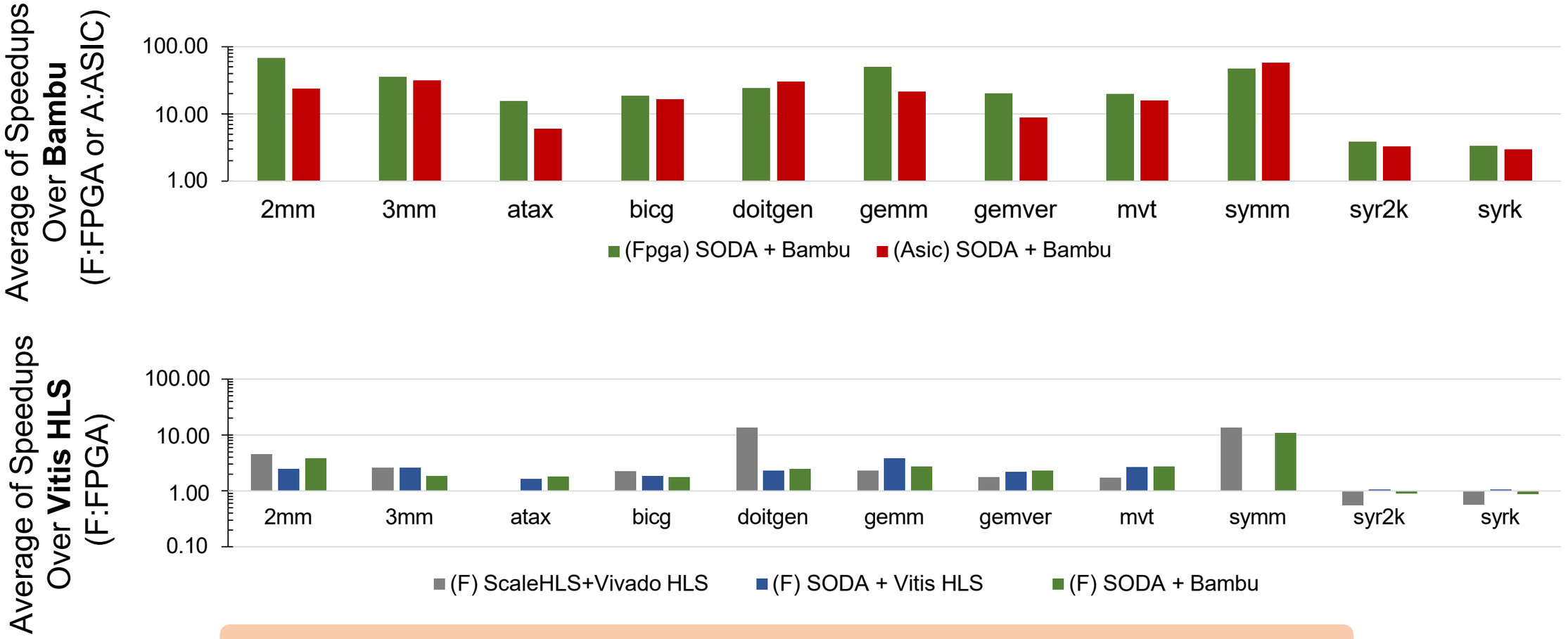
Benchmarking the SODA-OPT Pass Pipeline

SODA-OPT pass pipeline (For <HLS Tool>, Polybench Kernels) = Runtime



Kernels from: Louis-Noel Pouchet and Tomofumi Yuki. [n.d.]. PolyBench/C 4.2.1. http://polybench.sourceforge.net.

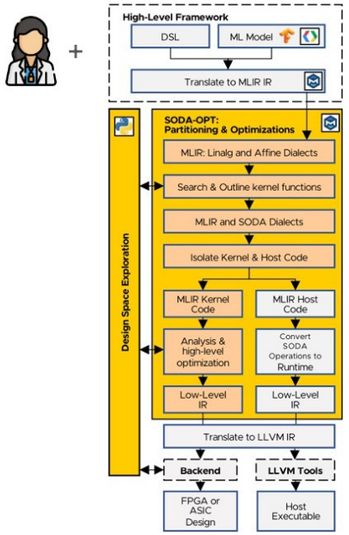
Performance Improvement vs Others



SODA-OPT derived accelerators **outperform**:

- **100%** of Bambu or Vitis HLS accelerators generated with no manual annotations
- **70%** of accelerators generated with current state of the art ScaleHLS

Contributions



- **An MLIR based** compiler flow for high-level synthesis
- An MLIR **dialect to search and outline** MLIR code at suitable abstractions
  - Kernels from HL applications can be outlined **at different granularities**
- Compiler **passes and pipelines** that enhance HLS of arbitrary regions of an application for **any target** (HLS Backend and Platform)
  - **Up to 60x speedup** over baseline designs that only leverage HLS optimizations
- Puts the domain scientist **in control** of custom accelerator generation with compiler passes. Transforming the scientist into a **Lead User** and potential **source of novel concepts**
- New HLS capabilities, essential to enable an "**agile hardware design**" approach

F1	F2	F3	F4	F5
Optimizations Without Manual Annotations	Automatic Partitioning of Host and Kernel Code	Design Space Exploration of Optimization Strategies	FPGA Support	ASIC Support

Reference

https://github.com/pnnl/soda-opt

https://hpc.pnl.gov/SODA/tutorials/2024/ISCA24.html