

# Tutorial Schedule

<b>9:00 – 9:30</b>	Antonino Tumeo	Agile Hardware Design for Complex Data Science Applications: Opportunities and Challenges.
<b>9:30 – 10:00</b>	Fabrizio Ferrandi	Bambu: An Open-Source Research Framework for the High-Level Synthesis of Complex Applications.
<b>10:00 – 10:30</b>	Antonino Tumeo	SODA-OPT: Enabling System-Level Design in MLIR for HLS and Beyond. Hands-on: From DNN Models to ASIC Devices with SODA-OPT
<b>10:30 – 11:00</b>		Coffee Break
<b>11:00 – 12:00</b>	Serena Curzel Michele Fiorito	Hands-on: Productive High-Level Synthesis with Bambu, Compiler Based Optimizations, Tuning and Customization of Generated Accelerators
<b>12:00 - 12:30</b>	Antonino Tumeo & Fabrizio Ferrandi	New features in SODA-OPT and Bambu



fast machine learning  
for science

# SODA-OPT

## Enabling System-Level Design in MLIR for High-Level Synthesis and Beyond

Tutorial: SODA Synthesizer: Accelerating Artificial Intelligence Applications with an End-to-End Silicon Compiler

September 1, 2025

**Antonino Tumeo**

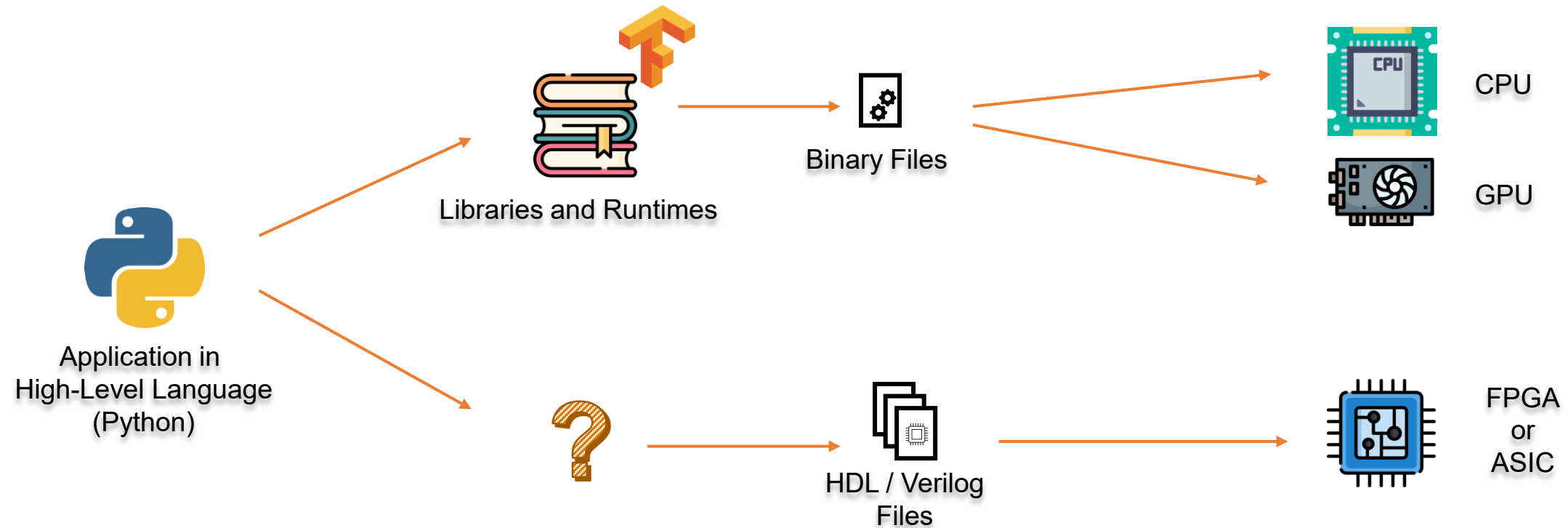
**Nicolas Bohm Agostini**



PNNL is operated by Battelle for the U.S. Department of Energy

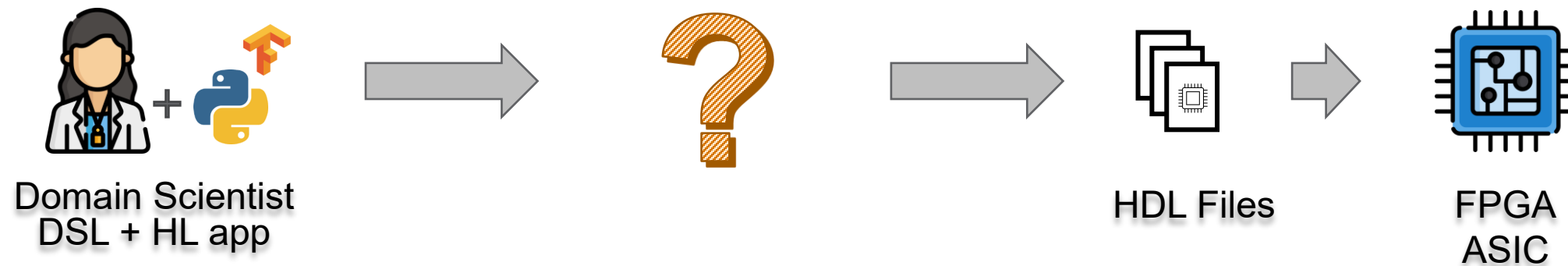


# Agile Hardware Design and Prototyping



# What is the 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?

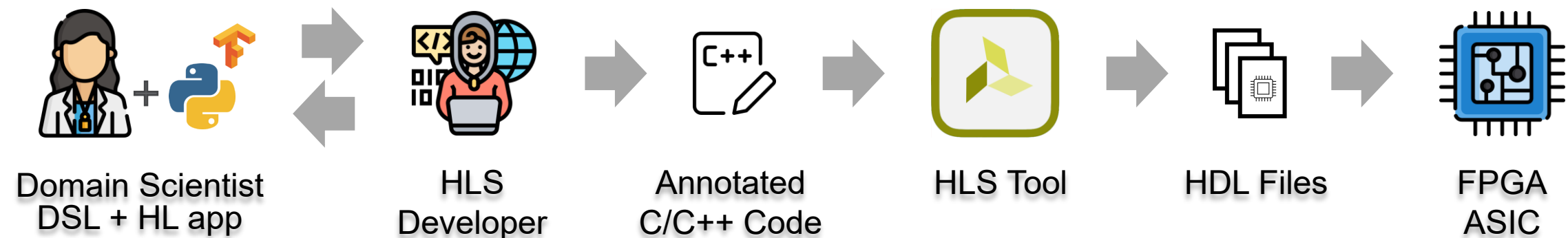


# How to address these challenges?

## Conventional approach : Ninja Programmer



## Conventional approach : HLS Developer



# 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**

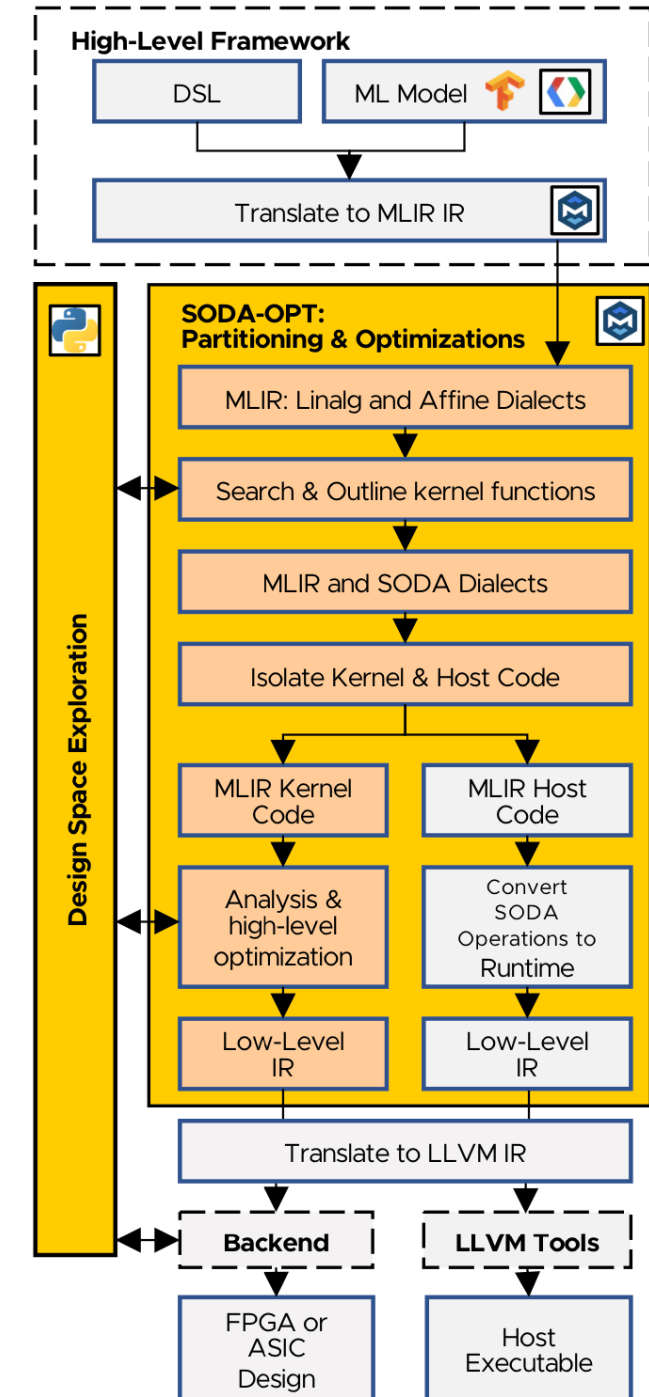
ASIC Support

# SODA-OPT

- Enables **MLIR-based inputs**
  - Supports any high-level application that can be converted into **linalg**, **affine** dialects
- Enables **System-Level Design**
- Enables **high-level optimizations** for the HLS backends
- Enables **DSE** of compiler options



+



# 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

```
%results:2 = "d.operation"(%arg0, %arg1) ({  
  // Regions belong to Ops and can have multiple blocks. Region  
  ^block(%argument: !d.type): Block  
    // Ops have function types (expressing mapping).  
    %value = "nested.operation"() ({  
      // Ops can contain nested regions. Region  
      "d.op"() : () -> ()  
    }) : () -> (!d.other_type)  
    "consume.value"(%value) : (!d.other_type) -> ()  
  ^other_block: Block  
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()  
  })  
  // Ops can have a list of attributes.  
  {attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

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

```

1 func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
2             %out: memref<f32>) {
3   linalg.dot ins(%A, %B: memref<100xf32>, memref<100xf32>)
4             outs(%out: memref<f32>)
5   return
6 }

```

## SCF abstraction

```

1 func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
2             %out: memref<f32>) {
3   %c0 = arith.constant 0 : index
4   %c100 = arith.constant 100 : index
5   %c1 = arith.constant 1 : index
6   scf.for %arg3 = %c0 to %c100 step %c1 {
7     %0 = memref.load %A[%arg3] : memref<100xf32>
8     %1 = memref.load %B[%arg3] : memref<100xf32>
9     %2 = memref.load %out[] : memref<f32>
10    %3 = arith.mulf %0, %1 : f32
11    %4 = arith.addf %2, %3 : f32
12    memref.store %4, %out[] : memref<f32>
13  }
14  return
15 }

```

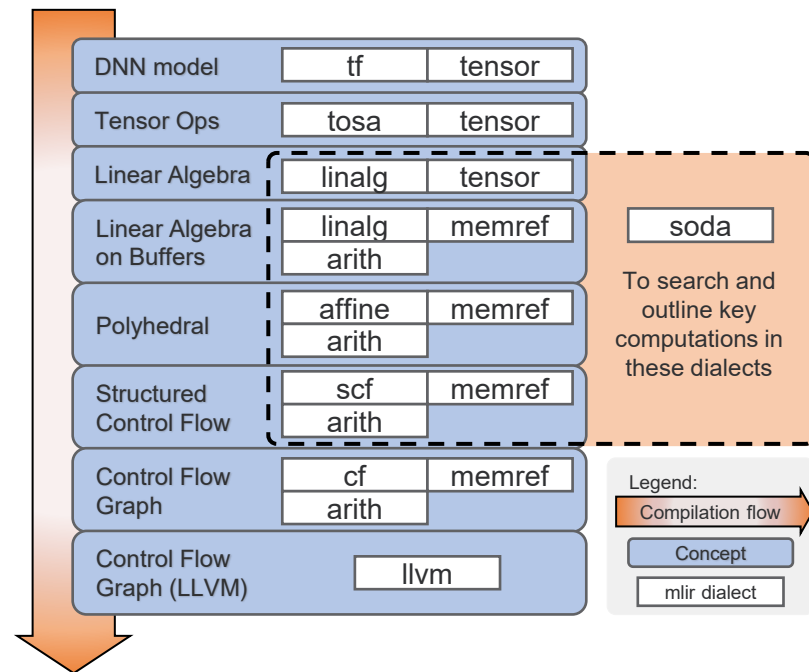
## CF abstraction

```

1 func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
2             %out: memref<f32>) {
3   %c0 = arith.constant 0 : index
4   %c100 = arith.constant 100 : index
5   %c1 = arith.constant 1 : index
6   cf.br ^bb1(%c0 : index)
7 ^bb1(%0: index): // 2 preds: ^bb0, ^bb2
8   %1 = arith.cmpi slt, %0, %c100 : index
9   cf.cond_br %1, ^bb2, ^bb3
10 ^bb2: // pred: ^bb1
11   %2 = memref.load %A[%0] : memref<100xf32>
12   %3 = memref.load %B[%0] : memref<100xf32>
13   %4 = memref.load %out[] : memref<f32>
14   %5 = arith.mulf %2, %3 : f32
15   %6 = arith.addf %4, %5 : f32
16   memref.store %6, %out[] : memref<f32>
17   %7 = arith.addi %0, %c1 : index
18   cf.br ^bb1(%7 : index)
19 ^bb3: // pred: ^bb1
20   return
21 }

```

# 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

```

1 module {
2   func.func @main(%A: memref<42x42xf32>, %B: memref<42x42xf32>,
3               %C: memref<42x42xf32>) {
4     soda.launch {
5       linalg.matmul ins(%A, %B : memref<42x42xf32>, memref<42x42xf32>)
6         outs(%C : memref<42x42xf32>)
7       soda.terminator
8     }
9     return
10  }}

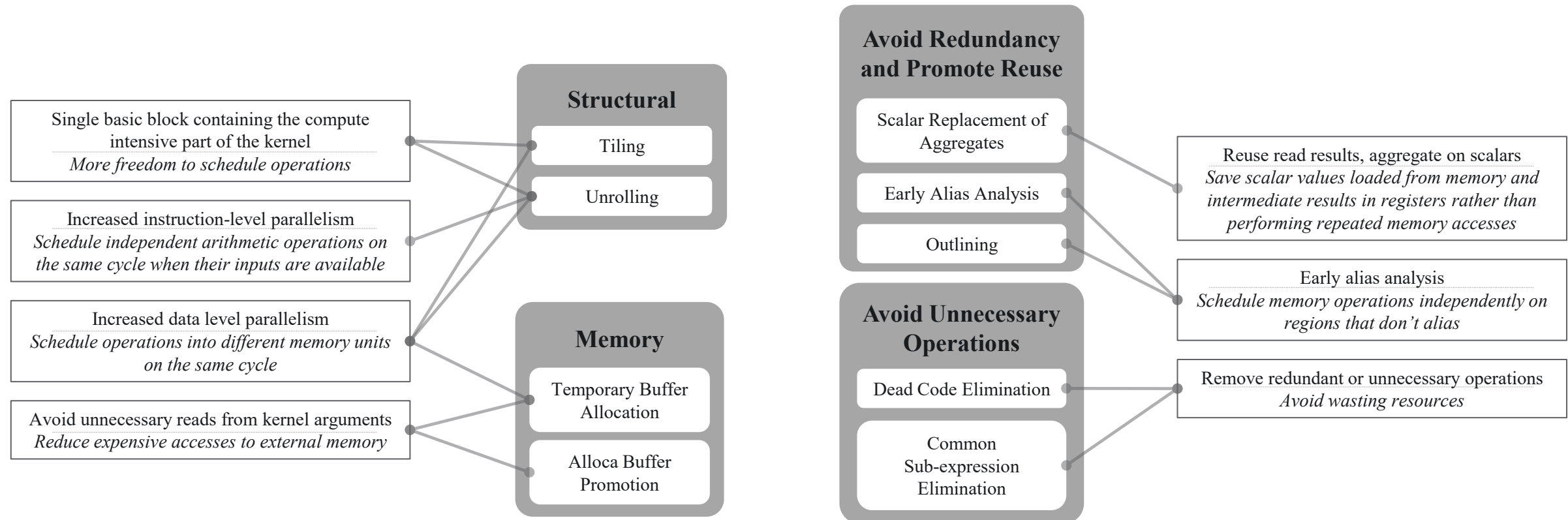
```

```

1 module attributes {soda.container_module} {
2   func.func @my_matmul(%A: memref<42x42xf32>, %B: memref<42x42xf32>,
3                   %C: memref<42x42xf32>) {
4     soda.launch_func @accelerators::@my_matmul_kernel args(
5       %A : memref<42x42xf32>, %B : memref<42x42xf32>, %C : memref<42x42xf32>)
6     return
7   }
8
9   soda.module @accelerators {
10    soda.func @my_matmul_kernel(%A_kernel: memref<42x42xf32>,
11                            %B_kernel: memref<42x42xf32>,
12                            %C_kernel: memref<42x42xf32>) kernel{
13      linalg.matmul ins(%A_kernel, %B_kernel : memref<42x42xf32>, memref<42x42xf32>)
14        outs(%C_kernel : memref<42x42xf32>)
15      soda.return
16    }}}

```

# Optimizations for High-Level Synthesis

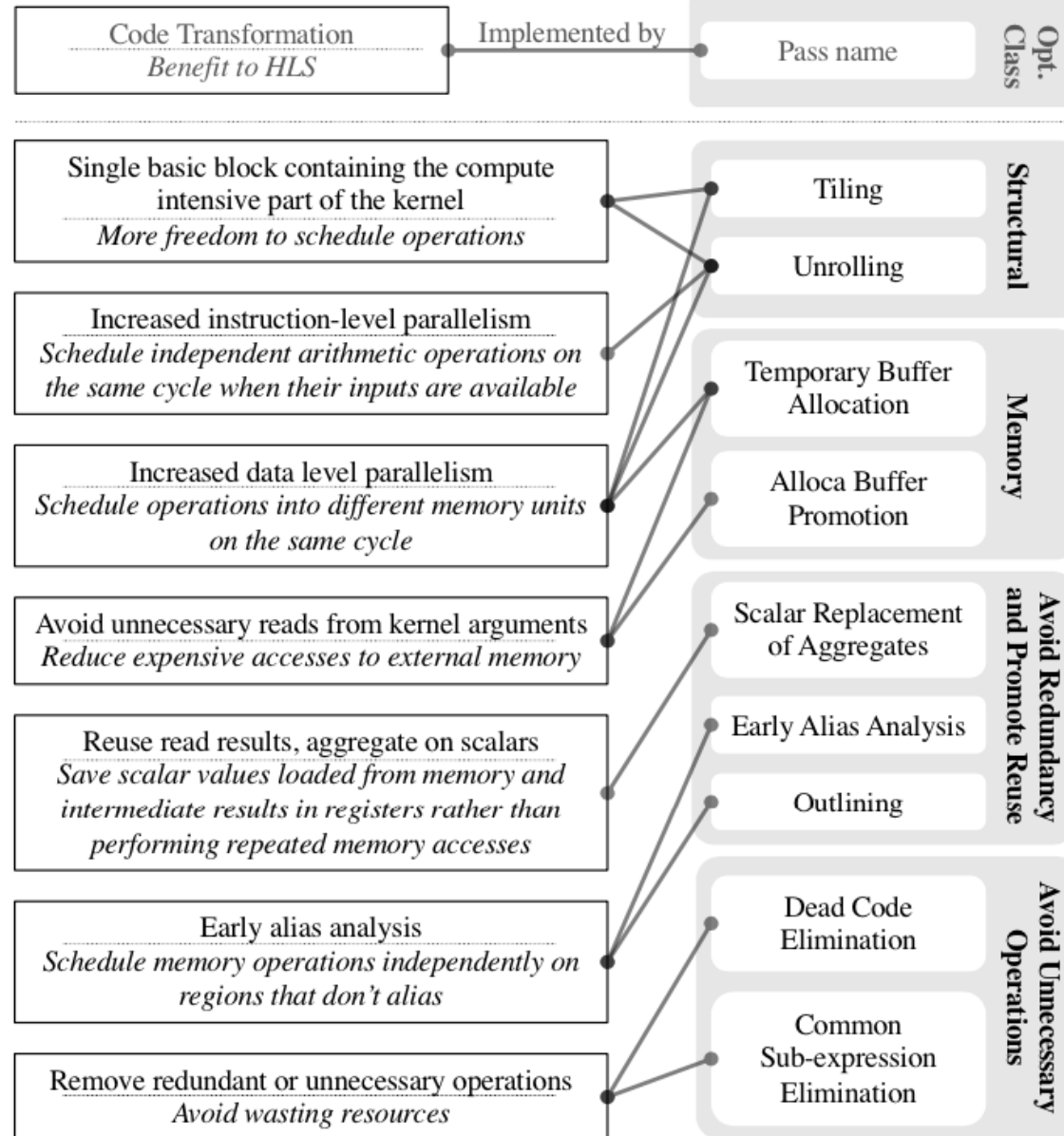


Legend:



# Current optimizations in SODA-OPT

Legend:



Generated with Clang -O2

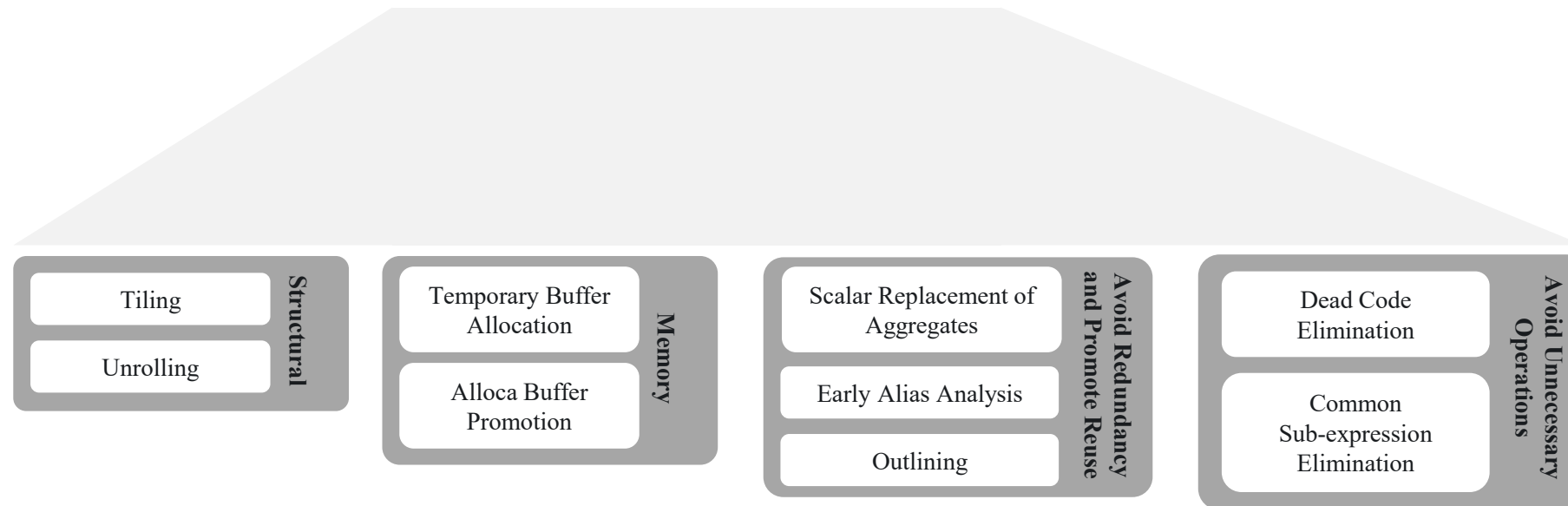
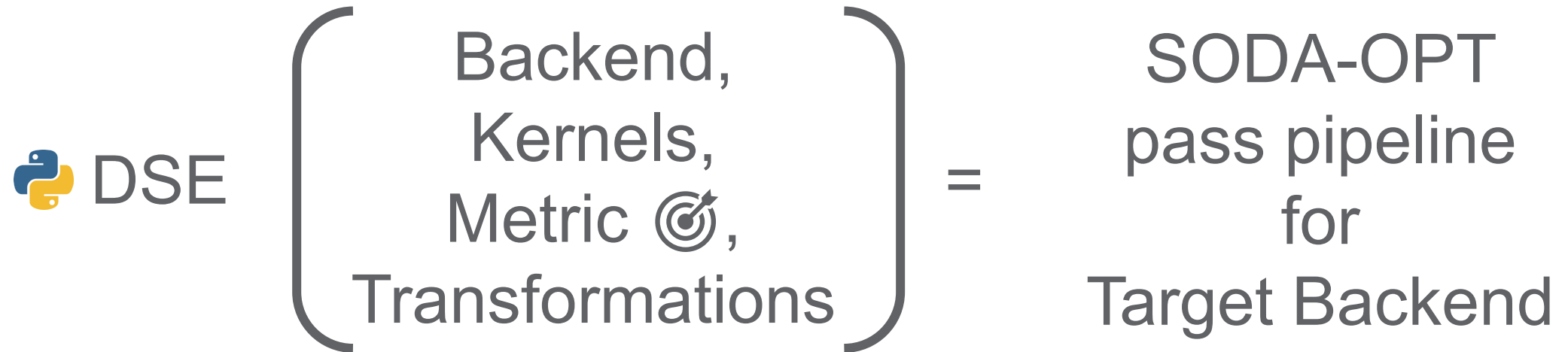
```
define void @matmul_kernel(
float* nocapture %0, float %1,
float* nocapture readonly %2,
float* nocapture readonly %3,
float %4) {
.preheader:
%5 = load float, float* %0, align 4
%6 = fmul float %5, %1
store float %6, float* %0, align 4
%7 = load float, float* %2, align 4
%8 = load float, float* %3, align 4
%9 = fmul float %7, %4
%10 = fmul float %9, %8
%11 = fadd float %6, %10
store float %11, float* %0, align 4
%12 = getelementptr float, float* %2, i64 1
%13 = load float, float* %12, align 4
%14 = getelementptr float, float* %3, i64 2
%15 = load float, float* %14, align 4
%16 = fmul float %13, %4
%17 = fmul float %16, %15
%18 = fadd float %11, %17
store float %18, float* %0, align 4
%19 = getelementptr float, float* %0, i64 1
%20 = load float, float* %19, align 4
%21 = fmul float %20, %1
store float %21, float* %19, align 4
%22 = load float, float* %2, align 4
%23 = getelementptr float, float* %3, i64 1
%24 = load float, float* %23, align 4
%25 = fmul float %22, %4
%26 = fmul float %25, %24
%27 = fadd float %21, %26
store float %27, float* %19, align 4
%28 = load float, float* %12, align 4
%29 = getelementptr float, float* %3, i64 3
%30 = load float, float* %29, align 4
%31 = fmul float %28, %4
%32 = fmul float %31, %30
%33 = fadd float %27, %32
%34 = fmul float %8, %1
%35 = fmul float %28, %22
%36 = fadd float %34, %35
%37 = fmul float %31, %26
%38 = fadd float %36, %37
%39 = fmul float %10, %1
%40 = fmul float %17, %4
%41 = fmul float %40, %20
%42 = fadd float %39, %41
%43 = fmul float %19, %4
%44 = fmul float %43, %24
%45 = fadd float %42, %44
%46 = fmul float %12, %1
%47 = fmul float %40, %22
%48 = fadd float %46, %47
%49 = fmul float %43, %26
%50 = fadd float %48, %49
store float %33, float* %0, align 4
store float %38, float* %7, align 4
store float %45, float* %9, align 4
store float %50, float* %11, align 4
ret void
}
```

Generated with SODA-OPT

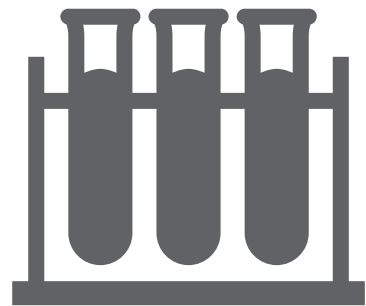
```
define void @matmul_kernel(
float* noalias nocapture %0, float %1,
float* noalias nocapture readonly %2,
float* noalias nocapture readonly %3,
float %4) {
%6 = load float, float* %0, align 4
%7 = getelementptr float, float* %0, i64 1
%8 = load float, float* %7, align 4
%9 = getelementptr float, float* %0, i64 2
%10 = load float, float* %9, align 4
%11 = getelementptr float, float* %0, i64 3
%12 = load float, float* %11, align 4
%13 = load float, float* %2, align 4
%14 = getelementptr float, float* %2, i64 1
%15 = load float, float* %14, align 4
%16 = getelementptr float, float* %2, i64 2
%17 = load float, float* %16, align 4
%18 = getelementptr float, float* %2, i64 3
%19 = load float, float* %18, align 4
%20 = load float, float* %3, align 4
%21 = getelementptr float, float* %3, i64 1
%22 = load float, float* %21, align 4
%23 = getelementptr float, float* %3, i64 2
%24 = load float, float* %23, align 4
%25 = getelementptr float, float* %3, i64 3
%26 = load float, float* %25, align 4
%27 = fmul float %6, %1
%28 = fmul float %13, %4
%29 = fmul float %28, %20
%30 = fadd float %27, %29
%31 = fmul float %15, %4
%32 = fmul float %31, %24
%33 = fadd float %30, %32
%34 = fmul float %8, %1
%35 = fmul float %28, %22
%36 = fadd float %34, %35
%37 = fmul float %31, %26
%38 = fadd float %36, %37
%39 = fmul float %10, %1
%40 = fmul float %17, %4
%41 = fmul float %40, %20
%42 = fadd float %39, %41
%43 = fmul float %19, %4
%44 = fmul float %43, %24
%45 = fadd float %42, %44
%46 = fmul float %12, %1
%47 = fmul float %40, %22
%48 = fadd float %46, %47
%49 = fmul float %43, %26
%50 = fadd float %48, %49
store float %33, float* %0, align 4
store float %38, float* %7, align 4
store float %45, float* %9, align 4
store float %50, float* %11, align 4
ret void
}
```

Operations	Clang -O2	SODA-OPT
Loads	20	12
Stores	12	4
Multiplications	20	16
Additions	8	8
Alias Analysis?	No	Yes
Scheduling Task?	Harder	Easier
Runtime (Cycles)	103	16
DSPs	8	16
LUTs	1929	2537

# DSE Engine

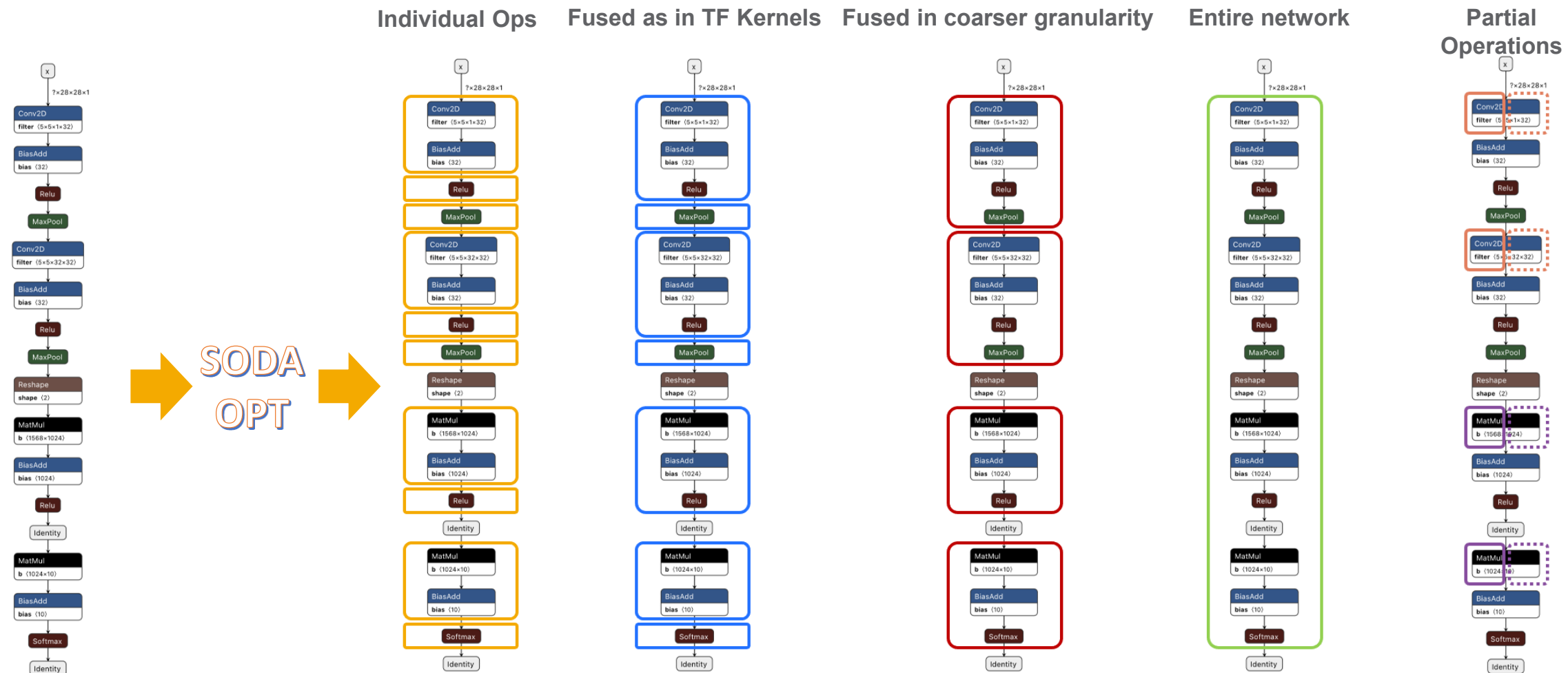


# Experiments and Evaluation



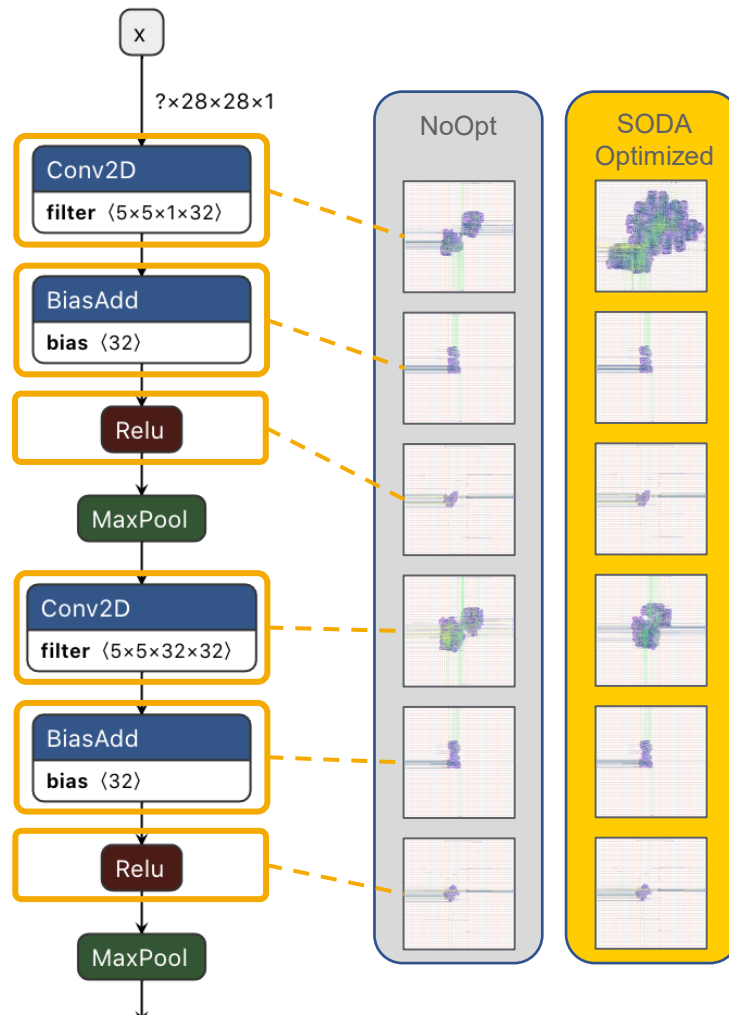
# Outlining in Different Granularities

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



# Tiling, Outlining and Optimizing

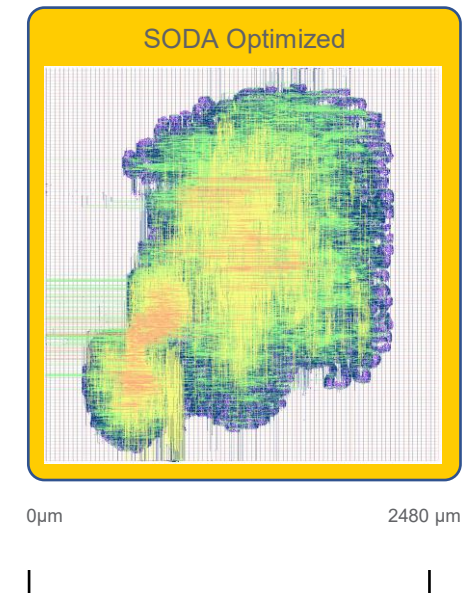
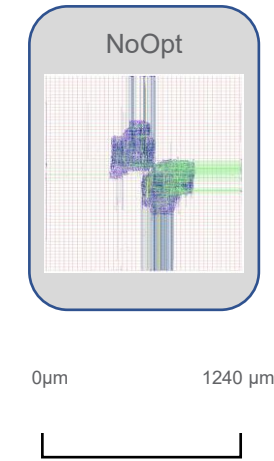
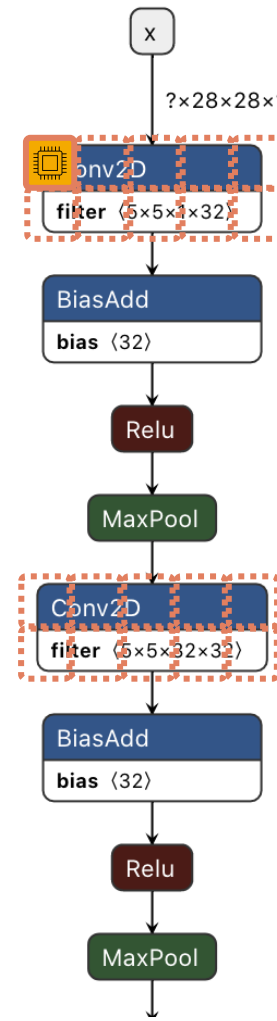
## Individual Ops



## Reusable Tile Accelerator

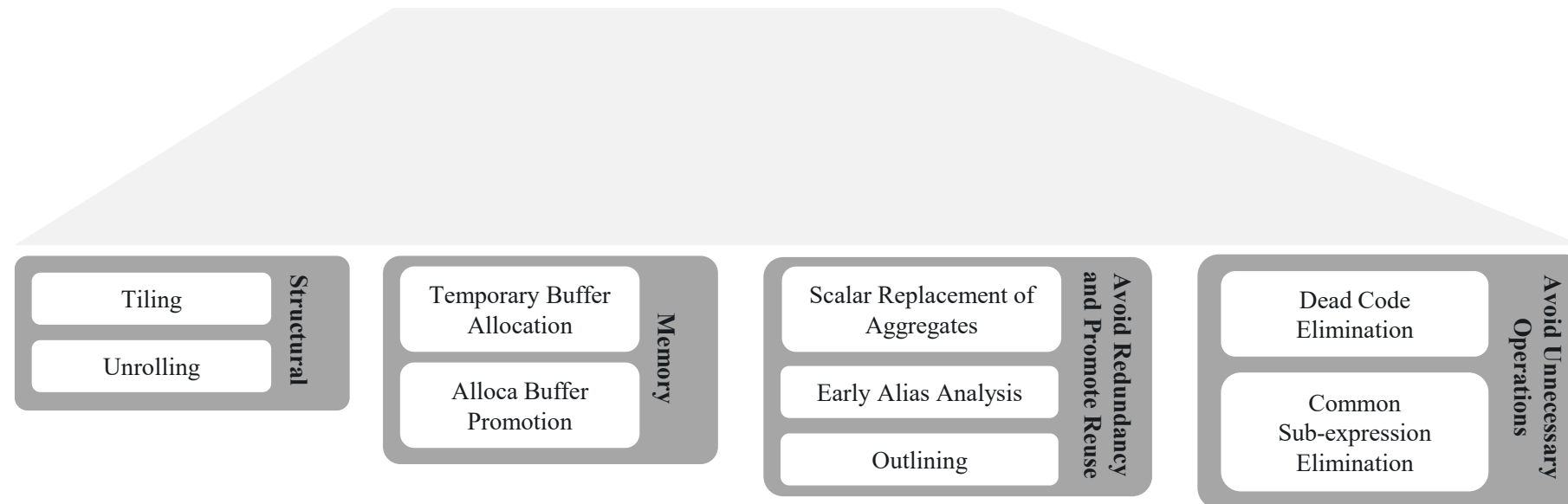
N,H,W,C,kH,kW,F  
1,1,14,8,1,1,1

- Careful selection of tile size enables accelerator reuse by multiple operators
- 4x the area, 15x speedup
- Automatically identified and generated

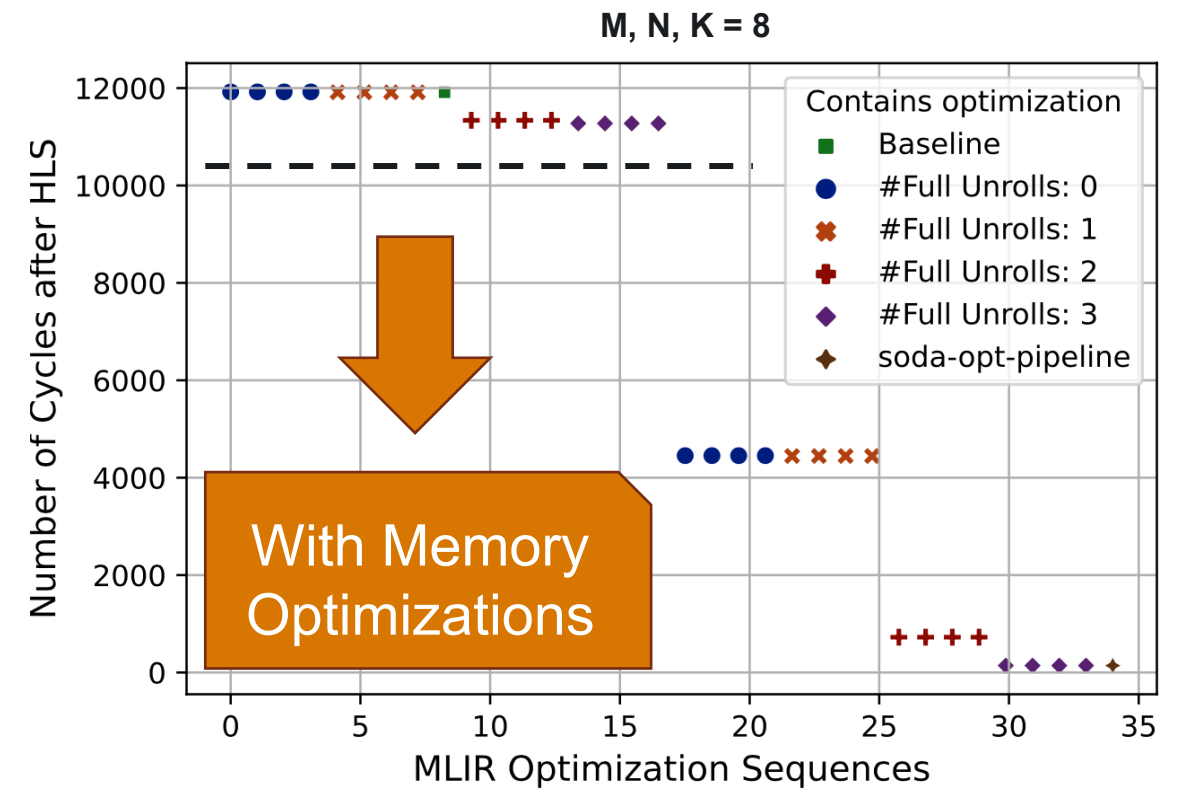
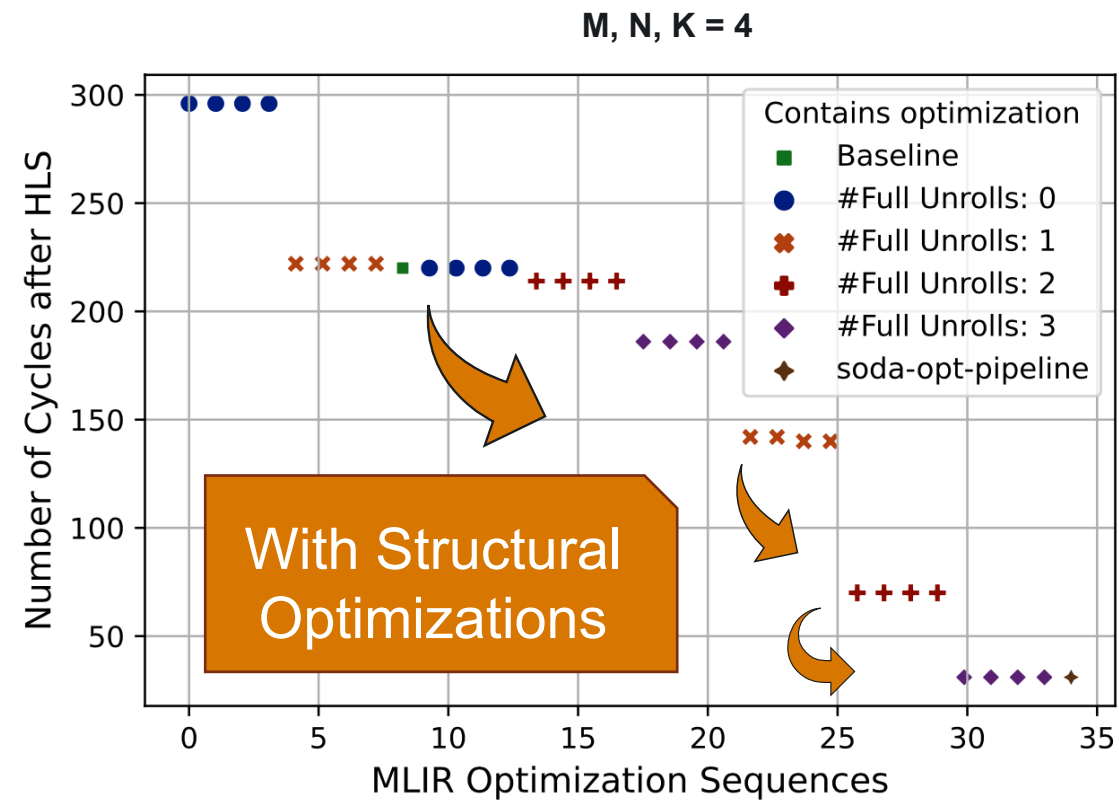


# Effects of DSE Engine

DSE  $\left( \begin{array}{c} \text{Bambu,} \\ \text{Kernels,} \\ \text{Runtime } \text{\textcircled{C}}, \\ \text{Transformations} \end{array} \right) = \text{SODA-OPT} \\ \text{pass pipeline} \\ \text{for Bambu}$

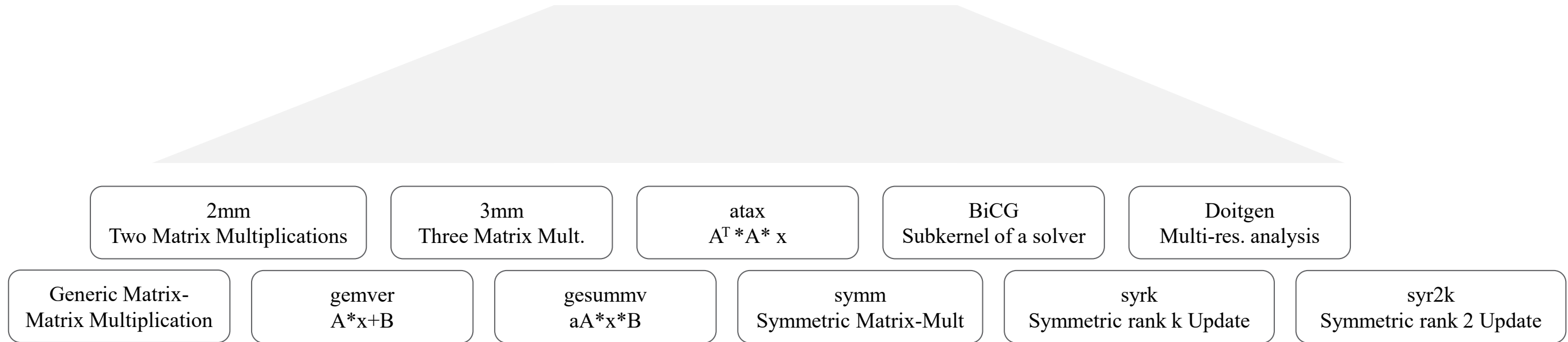


# Effects of DSE Engine on GEMM Kernels



# Benchmarking the SODA-OPT Pass Pipeline

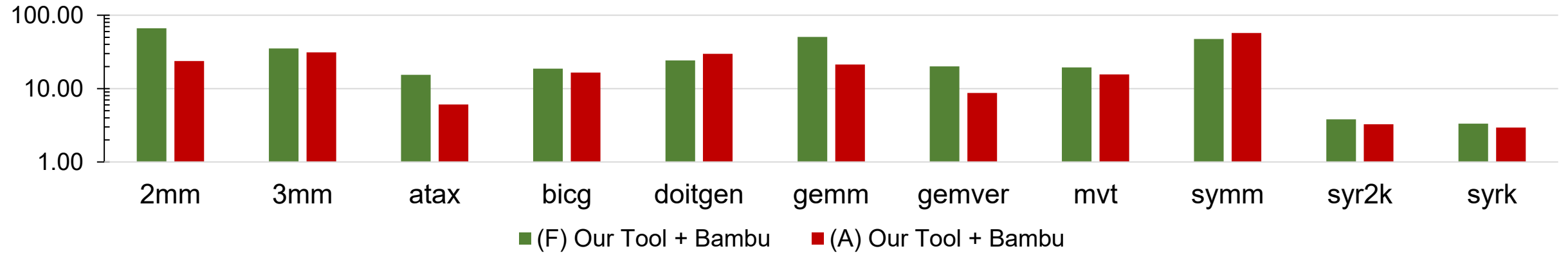
SODA-OPT pass pipeline  $\left( \begin{array}{c} \text{For } \langle \text{HLS Tool} \rangle, \\ \text{Polybench Kernels} \end{array} \right) = \text{Runtime} \text{ } \text{🎯}$



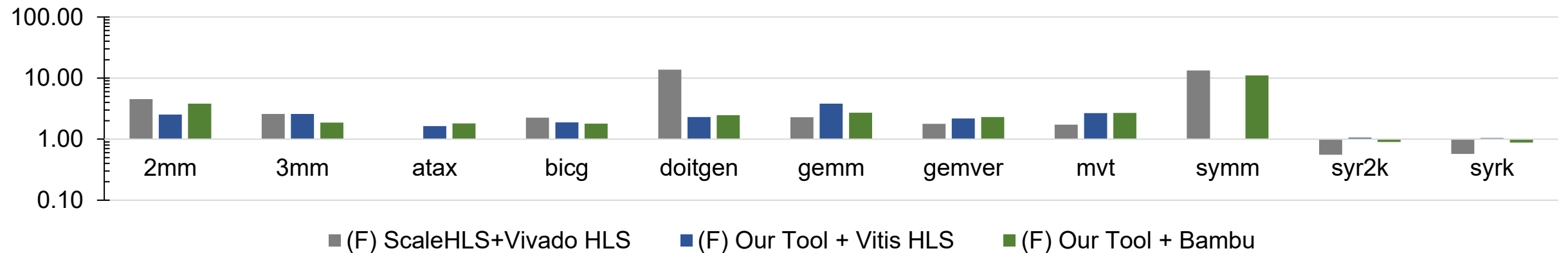
Kernels from: Louis-Noel Pouchet and Tomofumi Yuki. [n.d.]. PolyBench/C 4.2.1. <http://polybench.sourceforge.net>.

# Performance Improvement VS State of the Art

Average of Speedups  
Over **Bambu**  
(F:FPGA or A:ASIC)



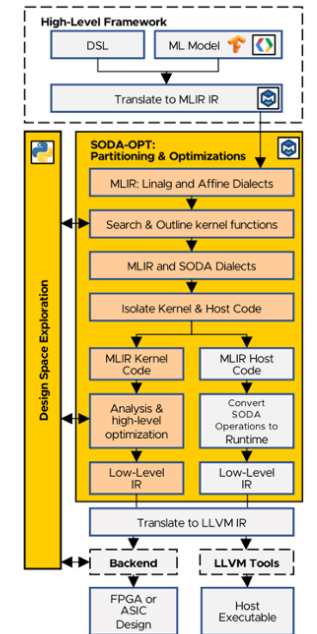
Average of Speedups  
Over **Vitis HLS**  
(F:FPGA)



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)
  - Our experiments show **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

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



Thank you!



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

