## Tutorial Schedule

**Morning:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Presenter</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 - 9:45am</td>
<td>Antonino Tumeo</td>
<td>Introduction and Agile Hardware Design for Complex Data Science Applications: Opportunities and Challenges.</td>
</tr>
<tr>
<td>10:30 - 11:00am</td>
<td></td>
<td>Break</td>
</tr>
<tr>
<td>11:00 - 11:45am</td>
<td>Antonino Tumeo</td>
<td>SODA-OPT: Enabling System-Level Design in MLIR for High-Level Synthesis and Beyond</td>
</tr>
<tr>
<td>11:45 - 12:00pm</td>
<td>Tumeo, Ferrandi</td>
<td>Preparation for the hands-on sessions</td>
</tr>
</tbody>
</table>

**Afternoon:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Presenter</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 - 2:15pm</td>
<td>Antonino Tumeo</td>
<td>Hands-on: From DNN Models to ASIC Devices with SODA-OPT</td>
</tr>
<tr>
<td>2:15 - 3:00pm</td>
<td>Fabrizio Ferrandi</td>
<td>Hands-on: Productive High-Level Synthesis with Bambu</td>
</tr>
<tr>
<td>3:00 - 3:30pm</td>
<td></td>
<td>Break</td>
</tr>
<tr>
<td>3:30 - 4:30pm</td>
<td>Fabrizio Ferrandi</td>
<td>Hands-on: Compiler Based Optimizations, Tuning and Customization of Generated Accelerators</td>
</tr>
<tr>
<td>4:30 - 5:30pm</td>
<td>Antonino Tumeo</td>
<td>High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics, Additional Features, and Closing remarks</td>
</tr>
</tbody>
</table>

All times reported are CET – Central European Time (CET+1).
SODA-OPT
Enabling System-Level Design in MLIR for High-Level Synthesis and Beyond

Tutorial: SODA Synthesizer
Accelerating Data Science Applications with an end-to-end Silicon Compiler

October 22, 2023
Antonino Tumeo
Chief Scientist
Agile Hardware Design and Prototyping

Application in High-Level Language (Python) → Libraries and Runtimes → Binary Files → CPU

Application in High-Level Language (Python) → Libraries and Runtimes → Binary Files → GPU

Application in High-Level Language (Python) → Libraries and Runtimes → Binary Files → FPGA or ASIC

Application in High-Level Language (Python) → Libraries and Runtimes → HDL / Verilog Files → FPGA or ASIC
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

1. Domain Scientist
   DSL + HL app
2. Ninja Programmer
3. HDL Files
4. FPGA/ASIC

Conventional approach: HLS Developer

1. Domain Scientist
   DSL + HL app
2. HLS Developer
3. Annotated C/C++ Code
4. HLS Tool
5. HDL Files
6. FPGA/ASIC
## Related Work

<table>
<thead>
<tr>
<th>Tool</th>
<th>Input</th>
<th>Backend tools</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScaleHLS</td>
<td>MLIR (Affine or Higher)</td>
<td>Vivado HLS</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CIRCT HLS</td>
<td>MLIR (Affine or Higher)</td>
<td>CIRCT</td>
<td>✓</td>
<td>X</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FROST</td>
<td>Halide, Tiramisu</td>
<td>Vivado HLS</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Hot &amp; Spicy</td>
<td>Annotated Python</td>
<td>SDSoc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>HeteroCL</td>
<td>Annotated C, OpenCL</td>
<td>Intel or Vitis HLS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HPVM2FPGA</td>
<td>HeteroC++</td>
<td>Intel HLS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Phism</td>
<td>C/C++</td>
<td>Vitis HLS</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

- **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

MLIR Dot Product Example and Lowering

Linalg abstraction

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

SCF abstraction

```c
1 func.func @dot(%A: memref<100xf32>, %B: memref<100xf32>,
2   %out: memref<f32>) { scf.for %arg3 = %c0 to %c100 step %c1 {
3   %2 = memref.load %A[2], %1 = %f32
4   %3 = memref.load %B[3], %1 = %f32
5   %4 = %3.arith.mulf %1, %f32
6   memref.store %4, %out[2] memref<f32>
7 return
8 }
```

CF abstraction

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

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

module attributes (soda.container_module) {
  func.func @my_mymul (%A: memref<42x42xf32>, %B: memref<42x42xf32>,
  %C: memref<42x42xf32>) {
    soda.launch_func @accelerators::@my_mymul_kernel args{
      %A : memref<42x42xf32>, %B : memref<42x42xf32>, %C : memref<42x42xf32>}
    return
  }
  soda.module @accelerators {
    soda.func @my_mymul_kernel(%A_kernel: memref<42x42xf32>,
    %B_kernel: memref<42x42xf32>,
    %C_kernel: memref<42x42xf32>) kernel{
      linalg.matmul (%A_kernel, %B_kernel : memref<42x42xf32>, memref<42x42xf32>)
      outs(%C_kernel : memref<42x42xf32>)
    }
    soda.return
  }
```
Optimizations for High-Level Synthesis

- Single basic block containing the compute intensive part of the kernel
  - More freedom to schedule operations
- Increased instruction-level parallelism
  - Schedule independent arithmetic operations on the same cycle when their inputs are available
- Increased data level parallelism
  - Schedule operations into different memory units on the same cycle
- Avoid unnecessary reads from kernel arguments
  - Reduce expensive accesses to external memory

**Structural**
- Tiling
- Unrolling

**Memory**
- Temporary Buffer Allocation
- Alloca Buffer Promotion

**Avoid Redundancy and Promote Reuse**
- Scalar Replacement of Aggregates
- Early Alias Analysis
- Outlining

**Avoid Unnecessary Operations**
- Dead Code Elimination
- Common Sub-expression Elimination

Legend:
- Code Transformation
  - Benefit to HLS
  - Implemented by
  - Pass name
  - Opt. Class

- Reuse read results, aggregate on scalars
  - Save scalar values loaded from memory and intermediate results in registers rather than performing repeated memory accesses
- Early alias analysis
  - Schedule memory operations independently on regions that don’t alias
- Remove redundant or unnecessary operations
  - Avoid wasting resources
Current optimizations in SODA-OPT

- Single basic block containing the compute intensive part of the kernel
- More freedom to schedule operations

Intrinsics

- Increased instruction-level parallelism
- Schedule independent arithmetic operations on the same cycle when their inputs are available

- Increased data level parallelism
- Schedule operations into different memory units on the same cycle

- Avoid unnecessary reads from kernel arguments
- Reduce expensive accesses to external memory

- Reuse read results, aggregate on scalars
- Save scalar values loaded from memory and intermediate results in registers rather than performing repeated memory accesses

- Early alias analysis
- Schedule memory operations independently on regions that don't alias

- Remove redundant or unnecessary operations
- Avoid wasting resources
DSE Engine

DSE = SODA-OPT pass pipeline for Target Backend
Experiments and Evaluation
Outlining in Different Granularities

- Present the impact of outlining and generating accelerators for different granularities of a DNN model
## Outlining in Different Granularities

<table>
<thead>
<tr>
<th>Layer type</th>
<th>Layer params</th>
<th>Individual Ops</th>
<th>Fused as in TF kernels</th>
<th>Fused in coarser granularity</th>
<th>Entire network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Baseline</td>
<td>Unrolled</td>
<td>Baseline</td>
<td>Unrolled</td>
</tr>
<tr>
<td>Conv2D</td>
<td>5x5,6,same</td>
<td>2,423,374</td>
<td>2,353,598</td>
<td>2,462,602</td>
<td>2,388,122</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2,526,225</td>
</tr>
<tr>
<td>Activation</td>
<td>ReLU</td>
<td>39,230</td>
<td>34,526</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AveragePooling2D</td>
<td>2x2, s2x2</td>
<td>88,244</td>
<td>84,338</td>
<td>88,244</td>
<td>84,338</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv2D</td>
<td>5x5,16,same</td>
<td>4,917,022</td>
<td>4,835,522</td>
<td>4,917,022</td>
<td>4,835,522</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5,175,970</td>
</tr>
<tr>
<td>Activation</td>
<td>ReLU</td>
<td>12,912</td>
<td>11,312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AveragePooling2D</td>
<td>2x2, s2x2</td>
<td>30,092</td>
<td>28,842</td>
<td>30,092</td>
<td>28,842</td>
</tr>
<tr>
<td>Dense</td>
<td>120 units</td>
<td>1,010,522</td>
<td>926,402</td>
<td>1,011,602</td>
<td>927,362</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense</td>
<td>84 units</td>
<td>213,446</td>
<td>195,722</td>
<td>214,202</td>
<td>196,394</td>
</tr>
<tr>
<td></td>
<td></td>
<td>214,202</td>
<td>196,394</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense</td>
<td>10 units</td>
<td>17,841</td>
<td>16,372</td>
<td>17,841</td>
<td>16,372</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense</td>
<td>4 units</td>
<td>454</td>
<td>410</td>
<td>454</td>
<td>410</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>8,754,773</td>
<td>8,488,476</td>
<td>8,742,059</td>
<td>8,477,362</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8,947,083</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**1.47x Speedup**
Tiling, Outlining and Optimizing

- 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

Bambu, Kernels, Runtime, Transformations

= SODA-OPT pass pipeline for Bambu

Structural

Tiling
Unrolling

Memory

Temporary Buffer Allocation
Alloca Buffer Promotion

Scalar Replacement of Aggregates
Early Alias Analysis
Outlining

Avoid Redundancy and Promote Reuse

Avoid Unnecessary Operations

Dead Code Elimination
Common Sub-expression Elimination
Effects of DSE Engine on GEMM Kernels

**M, N, K = 4**

- Number of Cycles after HLS
- MLIR Optimization Sequences
- Contains optimization:
  - Baseline
  - #Full Unrolls: 0
  - #Full Unrolls: 1
  - #Full Unrolls: 2
  - #Full Unrolls: 3
  - soda-opt-pipeline

With Structural Optimizations

**M, N, K = 8**

- Number of Cycles after HLS
- MLIR Optimization Sequences
- Contains optimization:
  - Baseline
  - #Full Unrolls: 0
  - #Full Unrolls: 1
  - #Full Unrolls: 2
  - #Full Unrolls: 3
  - soda-opt-pipeline

With Memory Optimizations
Benchmarking the SODA-OPT Pass Pipeline

SODA-OPT pass pipeline \( \begin{align*} &\text{For <HLS Tool>, Polybench Kernels} \\ = &\text{Runtime} \end{align*} \)

- 2mm: Two Matrix Multiplications
- 3mm: Three Matrix Mult.
- atax: \( A^T A^* x \)
- BiCG: Subkernel of a solver
- Doitgen: Multi-res. analysis

<table>
<thead>
<tr>
<th>Kernel Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Matrix-Matrix Multiplication</td>
<td>gemver A*x+B</td>
</tr>
<tr>
<td></td>
<td>gesummv aA<em>x</em>B</td>
</tr>
<tr>
<td>Symmetric Matrix-Mult</td>
<td>symm</td>
</tr>
<tr>
<td>Symmetric rank k Update</td>
<td>syrk</td>
</tr>
<tr>
<td>Symmetric rank 2 Update</td>
<td>syr2k</td>
</tr>
</tbody>
</table>

Performance Improvement VS State of the Art

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
Thank you!

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