Motivation

• Digital design tools and methodology have improved dramatically letting us create large SoCs with billions of transistors

• But completing these designs **(with software)**
  • Takes years
  • Costs hundreds of millions of dollars

Dozens of domain-specific processors or accelerators
  Machine Learning
  Image Processing
  Video Coding
  Cryptography
  Depth Processing

https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/2
Waterfall Approach to Accelerator Design

- A **waterfall** approach is still used for most accelerator designs

```
Application Analysis

ResNet MobileNet ...

Architectural Specification

RTL Design and Test

Physical Design

Software / Compiler Design
```
Agile Approach to Accelerator Design

- We explore an agile hardware/software design flow
  - Incrementally update the hardware accelerator and software to map to it

```
<table>
<thead>
<tr>
<th>Base Hardware</th>
<th>Compiler Toolchain v0</th>
<th>Base Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator v0</td>
<td>Application 1</td>
<td>Accelerator v1</td>
</tr>
<tr>
<td></td>
<td>Application 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Incremental</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Updates</td>
<td></td>
</tr>
</tbody>
</table>
```

Power, Performance, Area
1. Accelerator must be **configurable**
   - So we can map new or modified applications to it (although with lower efficiency)

2. Hardware and compiler must **evolve together**
   - Any change in hardware must propagate to compiler automatically

---

**Agile Approach to Accelerator Design**

- **Base Hardware**
  - **Accelerator v0**
  - **Compiler Toolchain v0**

- **Application 1**
- **Application 2**

- **Power, Performance, Area**

- **Incremental Updates**
  - **Base Hardware**
  - **Accelerator v1**
  - **Compiler Toolchain v1**
SoC with a Coarse-Grained Reconfigurable Array

- Our accelerator is an island-style CGRA
  - Processing element (PE) tiles – potentially heterogeneous
  - Memory (MEM) tiles
  - Statically configured interconnect
- Programmable, but allows exploiting parallelism and locality
Halide Program for a 3x3 Convolution

Algorithm:
RDom r(0, 3, 0, 3);
output(x, y) += input(x + r.x, y + r.y)
   * weight(r.x, r.y)

Schedule:
input.in().store_at(output, y)
   .compute_at(output, x);
output.accelerate({input}, y);
output.unroll(r.x).unroll(r.y);
Software Compiler

Application Halide Program → Lower → CoreIR Graph

Unified Buffer

input → Mul → Add

Mul → Add → Mul

CoreIR Dataflow Graph

CPU Code

... Add → output
Software Compiler

- Application Halide Program → CPU Code
- Lower → CoreIR Graph
- Map PE and Memory → Mapped CoreIR Graph
- CGRA Place & Route → CGRA Bitstream

Deep Dive Talk 1: Connecting Polyhedral Optimization to CGRA Buffer Generation

Diagram:
- Input
- Mapped Memory
- Shift Registers
- Mapped Kernel
- Mapped CoreIR Graph
- Output
Software compiler must evolve with hardware!

- **Application Halide Program**
  - Lower
  - CoreIR Graph
  - Map PE and Memory
    - Mapped CoreIR Graph
    - CGRA Place & Route
      - CGRA Bitstream

- **Hardware independent**
  - Depends on the PE and Memory hardware
  - Depends on the interconnect hardware
Our Key Contribution

• Traditionally, designers create parameterized hardware generators that communicate with the software compiler through configuration files

• We create mini languages whose semantics are sufficiently expressive to communicate both configuration values and how changes to those values impact other layers in the system

• Our system has three mini-languages or domain-specific languages (DSLs)
  • PEAk for PEs, Lake for memories, Canal for interconnect
Our DSL-based Hardware Generation and Software Compilation Flow

PEak Program (PE spec) → PEak Compiler

PEak Compiler → PE HW in Magma → Magma Compiler

Magma Compiler → CGRA Verilog

Halide Program → Halide Compiler

Halide Compiler → CoreIR Graph

CoreIR Graph → PE and MEM Mapper

PE and MEM Mapper → Mapped CoreIR Graph → Place & Route Engine

Place & Route Engine → CGRA Bitstream

Deep Dive Talk 2: A General Mapping Flow in an Agile Hardware World
Our DSL-based Hardware Generation and Software Compilation Flow

Deep Dive Talk 4: Lake Memory Generator and SMT for Automated Memory Configuration
Our DSL-based Hardware Generation and Software Compilation Flow

Deep Dive Talk 3: Formal Checkers and Solvers for Hardware Design and Verification
Our DSL-based Hardware Generation and Software Compilation Flow

Deep Dive Talk 5: Design Space Exploration of Processing Element Architectures
Summary

• Domain-specific architectures that are specialized yet somewhat programmable
  • CGRAs specialized for different domains

• Compiler that compiles high-level programs in a domain-specific language to our CGRA

• Start with a simple CGRA and create a working system, then make incremental changes to it – creating an agile design flow

• As we incrementally evolve our CGRA hardware for changing applications, our compiler tracks the hardware changes

• Design space exploration framework on top of our flow that let’s you easily find the best CGRA architecture for your domain of interest