AHA (Agile Hardware): Progress & Plans

Clark Barrett, Kayvon Fatahalian, Pat Hanrahan,
Mark Horowitz, Priyanka Raina

1/29/2019

Stanford | ENGINEERING
Electrical Engineering

Team

Pat Hanrahan

Clark Barrett

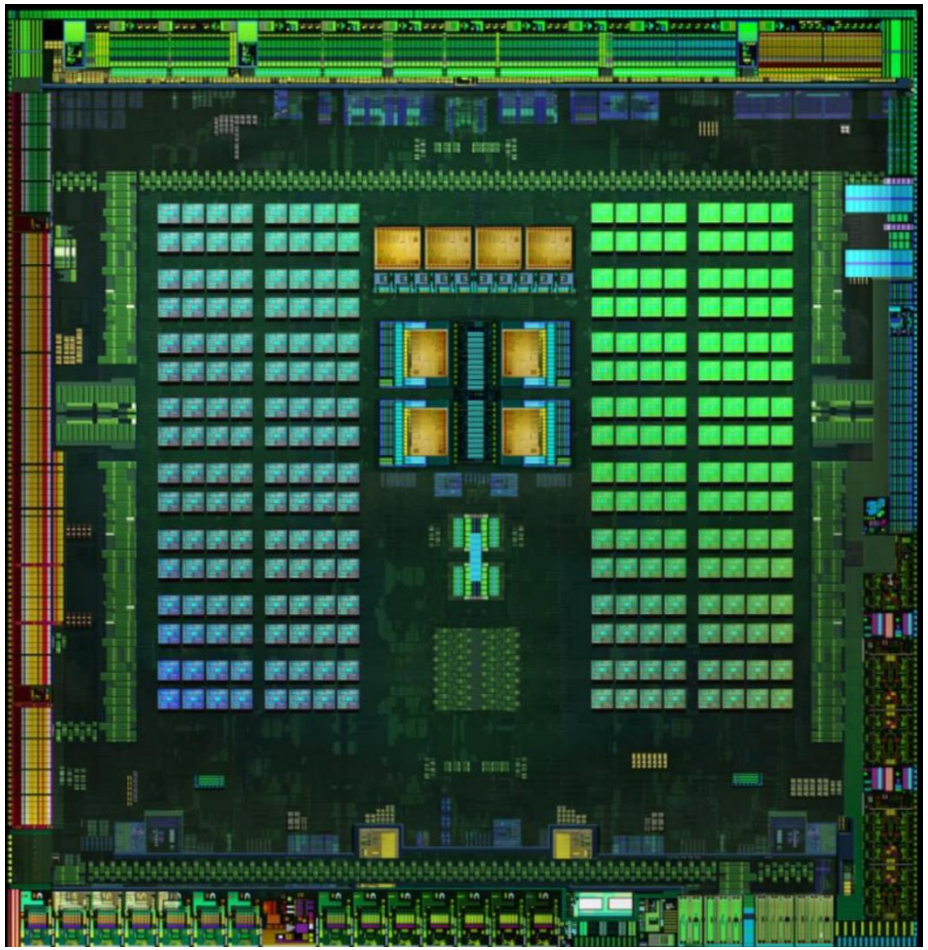
Kayvon Fatahalian

Priyanka Raina

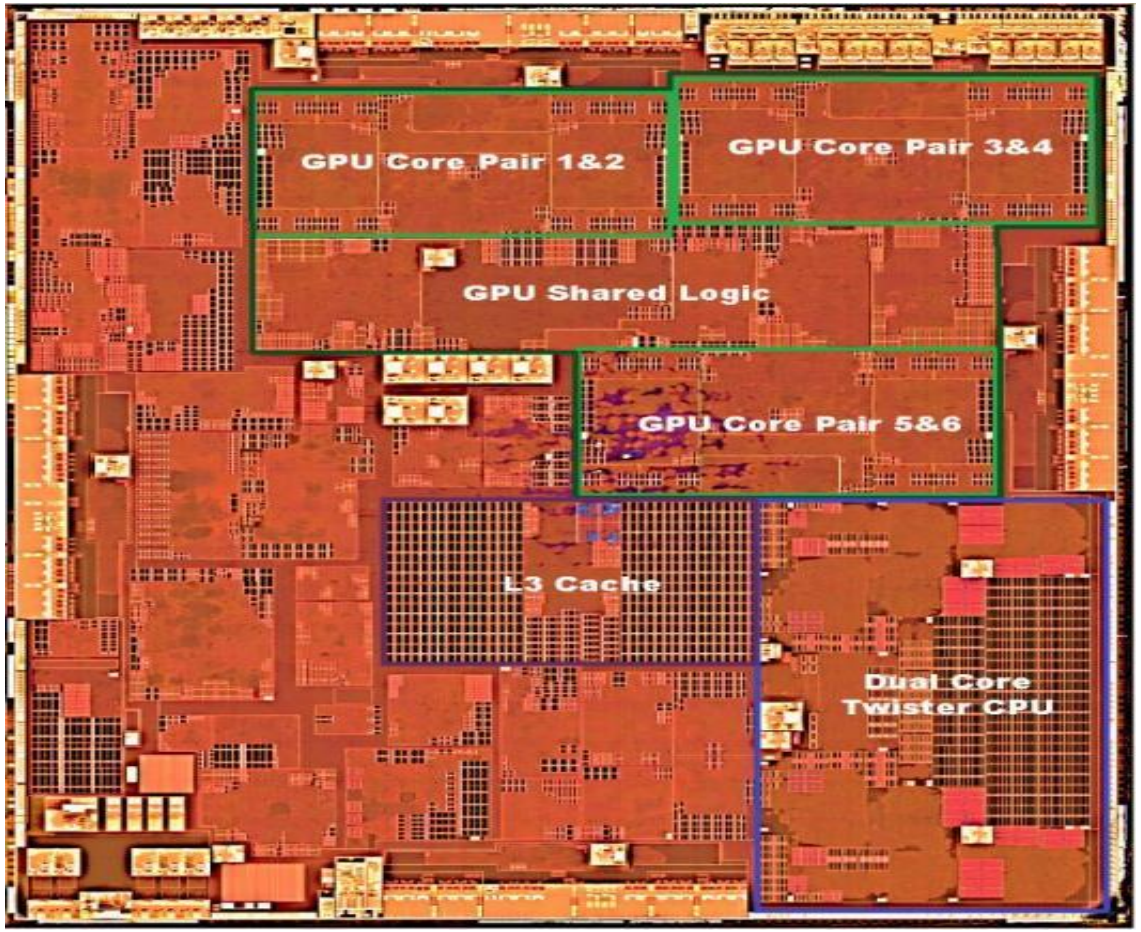
The Semiconductor Business is Changing

- Scaling is slowing
- Chip design is very expensive
- The industry is consolidating
- Good VLSI designers are now all going to systems companies
  - The industry is re-integrating

Specialized Hardware - DSSoC



Tegra X1



Apple A9

The New Challenge:

- How to efficiently create these systems w/ accelerators
  - Which is **much** harder than producing the accelerator

- Porting an application to hardware
  - Is about reframing the algorithm as much as hardware design
  - Need expertise in the application area and hardware design

AHA – Agile Hardware

It is partly a mindset, but it is more than that

- Start with application
- Quickly map to hardware
- Repeat

Success Depends On Avoiding:

- Specialized Hardware Mistake
- Save Computing Mistake

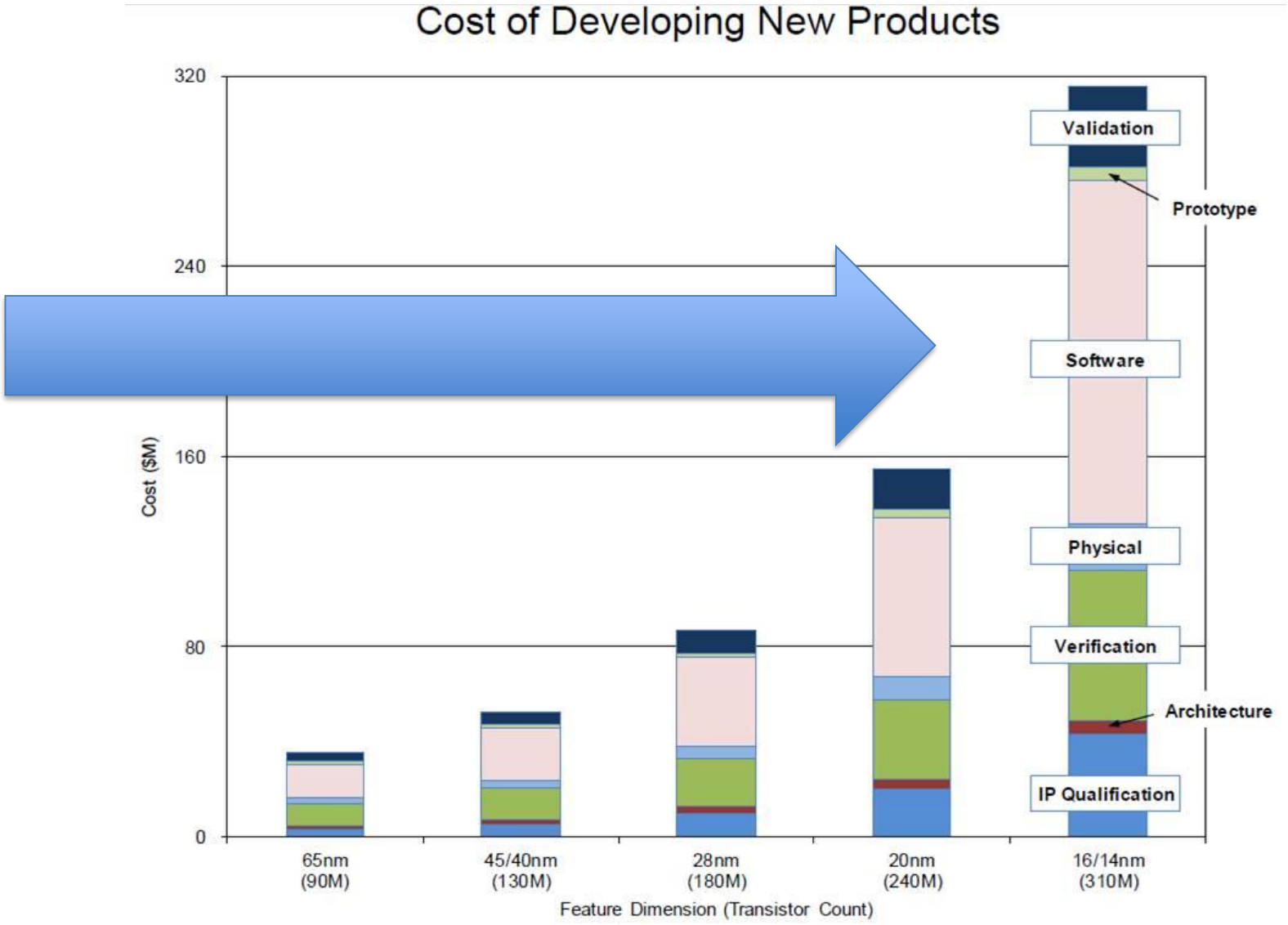
The Specialized Hardware Mistake

- Study the application
- Build the hardware
- Then write the software for the machine
  - And that is the only application this hardware will ever support…
- Repeat

This Is The

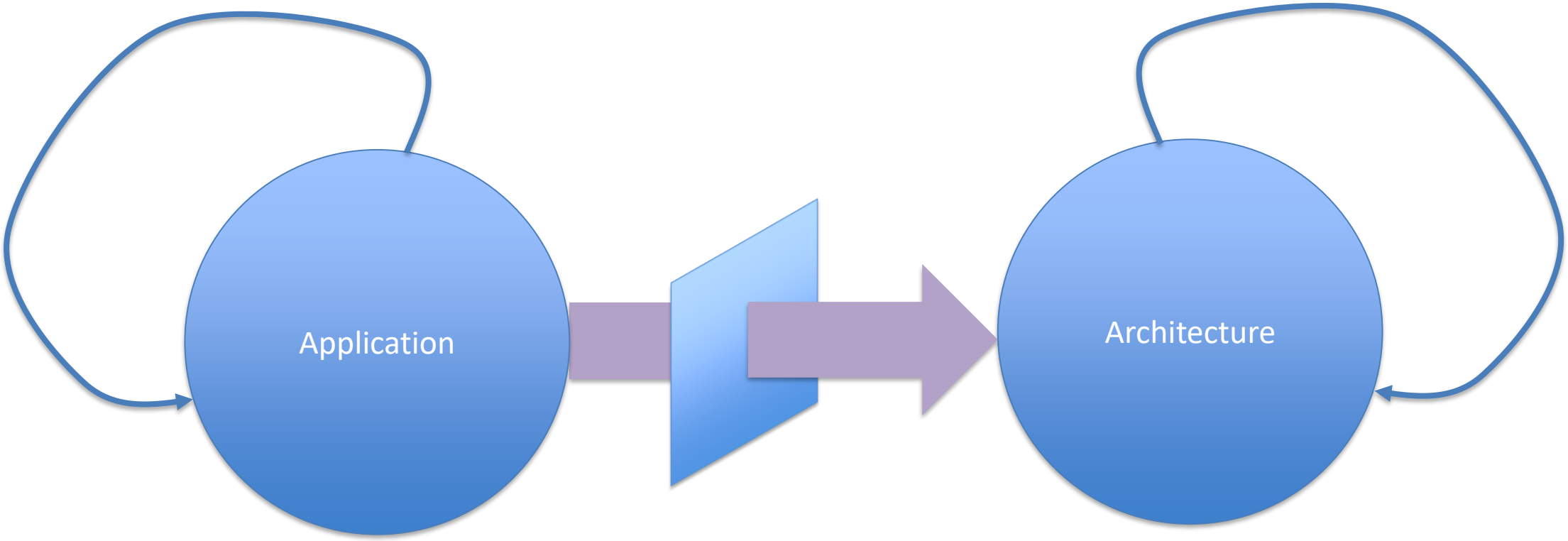
Model of Design

Doesn’t Address The Biggest Issue



Source: IBS

To Quickly Map App to Hardware



Need Some Isolation

The Abstract Hardware Model
(Avoiding the Save Computing Mistake)

- Accelerators only work for local, compute intensive, applications
  - Everything else will be limited by the memory system

- Our hardware IR should be good for these applications

Optimizing with an “ISA”

An Agile Approach to Accelerators

- Key idea:
  - Maintain end-to-end flow no matter what
  - Always be able to compile new applications
- Incrementally change tools/fabric to improve performance
- Accelerator becomes optimization of underlying fabric
  - Heterogeneous fabric, which may or may not need “new” instructions

To Quickly Map App to Hardware

- Application needs to be written in a DSL (Visual Computing)
  - Halide
- Compiler maps it to a programmable hardware model (ISA)
  - CoreIR
- Map, place/route lowered application to target hardware
  - CGRA

Why Halide

- Separates schedule from function

- Already is an embedded language
  - Easy to indicate what should be accelerated
  - Generates good code for the rest

Why CoreIR

- Formal semantics
  - Facilitates formal checking of design flow
- Representation (DAGs of operations)
  - Matches the representation of our applications
- Allows linking of inputs from other systems
  - It isn’t ego-centric

Why CGRA

- It is universal
  - It can map any application
- It is a spatial architecture
  - So we can make it look like any accelerator

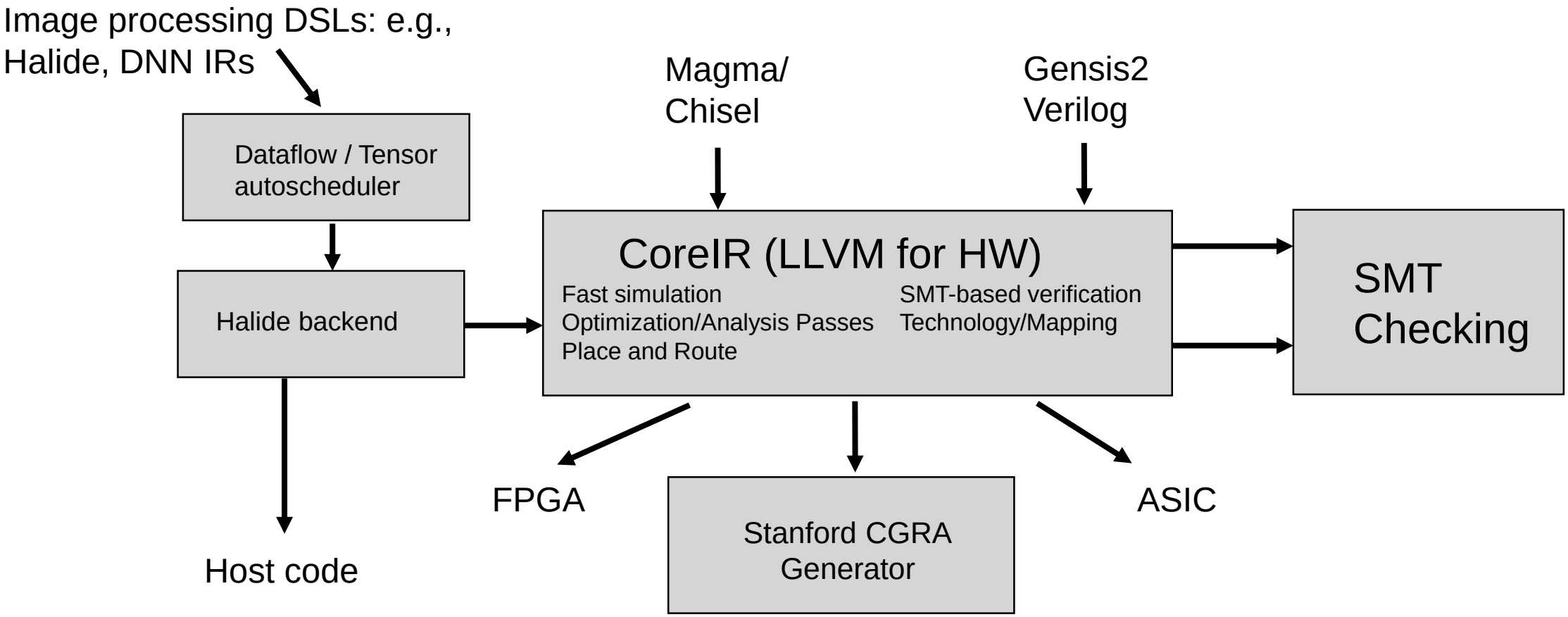
Also Need To Build The Hardware Target

- This is an SoC with a CGRA
  - Where the CGRA is continually being updated
  - These updates improve the “accelerator”
- Need to create a generator to create this SoC

Progress and Plans

- Application to coreIR
  - Halide compiler work, autoscheduling
- CoreIR, and mapping coreIR to CGRA
- CGRA design, generating CGRA
- Test and validation environment

AHA Toolchain (working)



Halide to CoreIR

- Newly implemented codegen capabilities
  - ROM generation
  - multi-dimensional linebuffers
  - valid bit generation from linebuffers

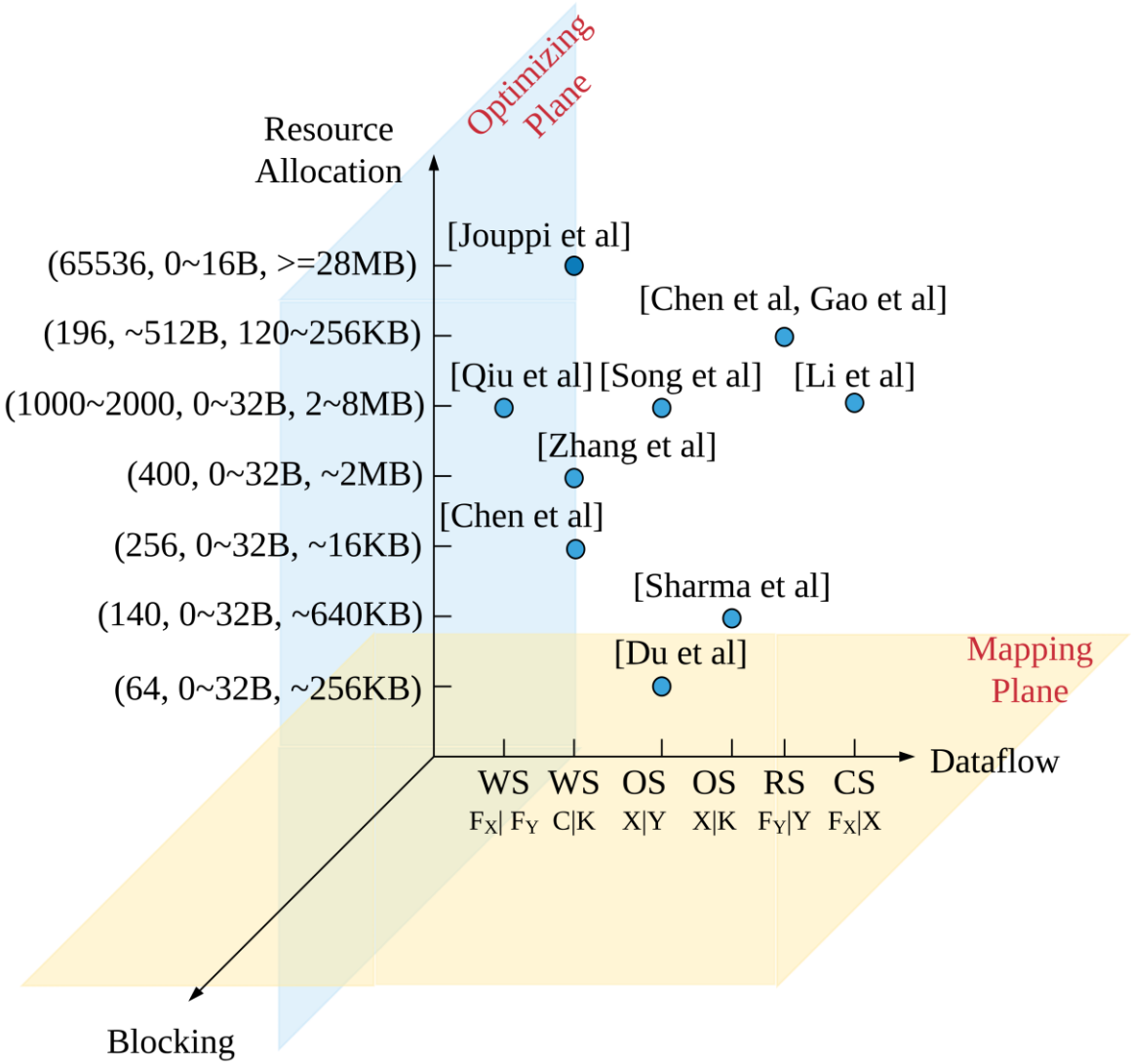
- Generating hardware from more applications
  - Unit tests: logical right shift, counter, …
  - Full apps: demosaic, optical flow, camera pipeline, …

Halide to CoreIR Plans

- Currently rebasing to top-of-tree
  - First application complete: 3x3 convolution with updated compiler
- Re-expressing scheduling primitives
  - Hardware accelerator tiling based on GPU tiling
  - Primitives for linebuffer and full pipeline throughput
- Merge DNN and image hardware branches to top-of-tree

Halide Auto-schedulers

- Auto-schedulers
  - Both for Halide and for Halide like IR (joint work with Google)
- Complete scheduler for DNNs
  - Can use any existing hardware



Auto-scheduler Plans

- Continue work on g.p. Halide scheduler
  - Joint work with Andrew Adams, Facebook
- Create auto-scheduler for joint hardware generation
  - Streaming and blocking computations

Addressing the Hard Problem:

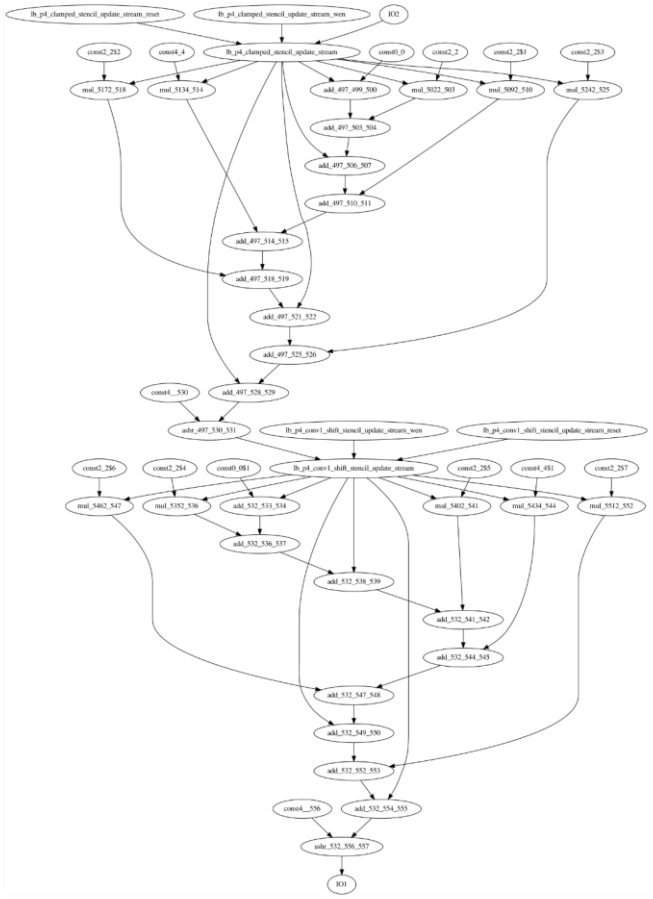
- No one wants hardware for this application
  - They want hardware for this type of application
- How to create the hardware they want?
  - CGRA might be a solution
- Look at creating generalize functions
  - Explore different implementation options

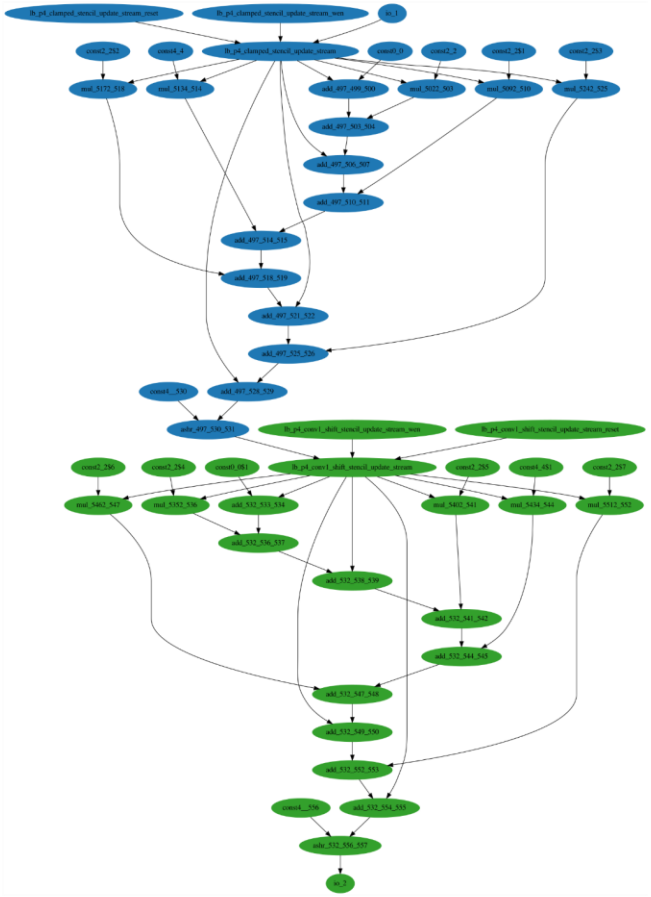
CoreIR to CGRA

- Basic system is up and running
  - Can read inputs from Verilog, FIRRTL, Rigel, Halide, Magma
- Formal semantics
  - Primitives have equivalent to SMT bit vectors
- Created new P&R flow
  - It is in use in current flow
- Extensible framework
  - Easy to add passes or extentions to the system

Kernel Discovery from Dataflow Graph

- Two convolutions
- Two kernels

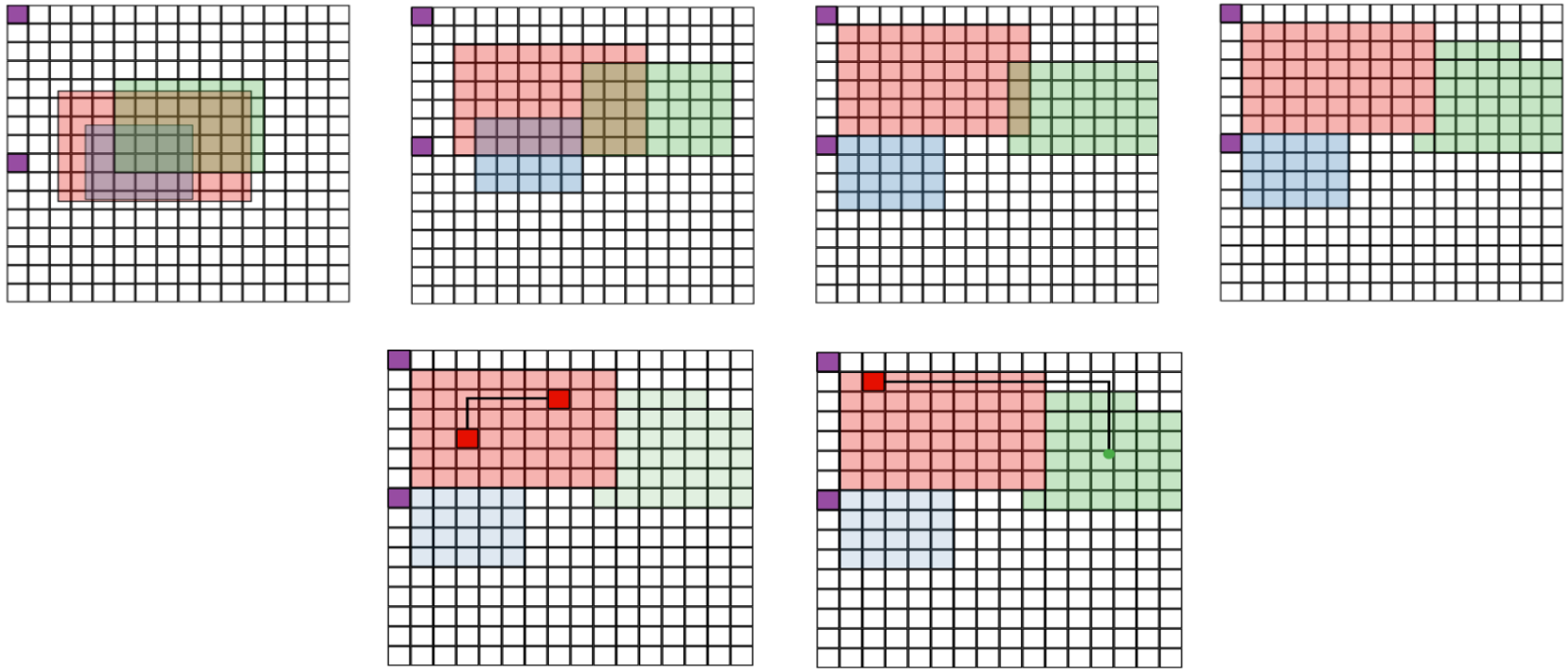




Cascade Example

Placement Details

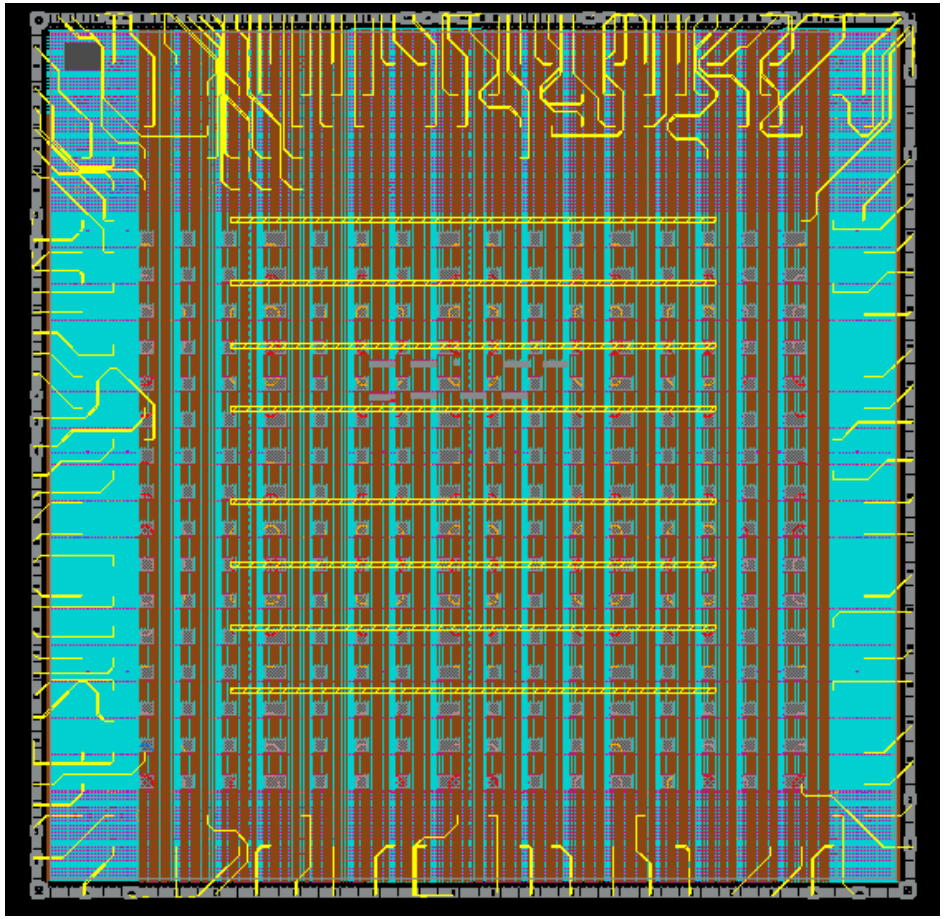
- Force-Directed Global Placement
- Simulated Annealing Detailed Placement

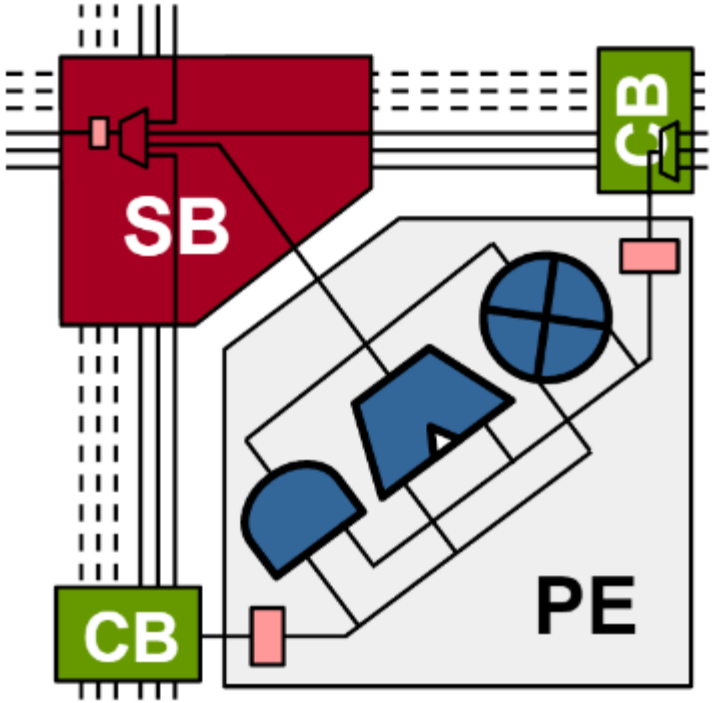


CoreIR to CGRA Plans

- Extend coreIR feature set to enable wider set of applications
  - More operand types (e.g. floating point)
  - More operators (div, generalized exponentials)
- Extended coreIR and P&R analysis passes:
  - Register retiming
  - Improved mapping/packing pass for CGRA
- Suggest changes to CGRA design to improve efficiency

Taped Out Our First CGRA



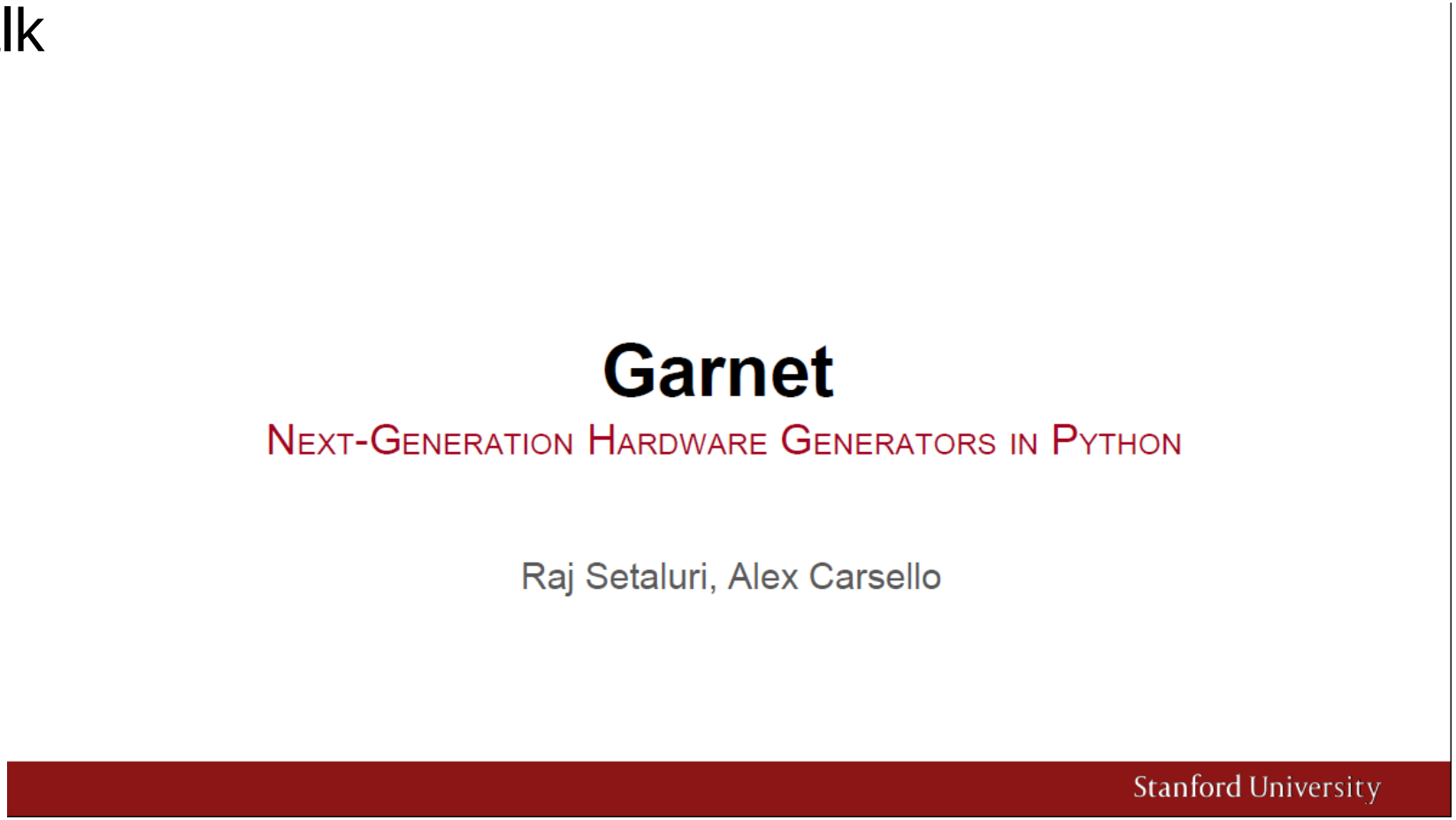


CGRA Plans

- Fix problems with current system to make it easily buildable
  - Create virtual tape-out flow
- Increase capability of the FU
  - Bfloat16 data types, divide, transcendental support
- Memory hierarchies
  - Fast reconfiguration, testing and debug support
- Explore supporting dynamic launch, as used in modem processing

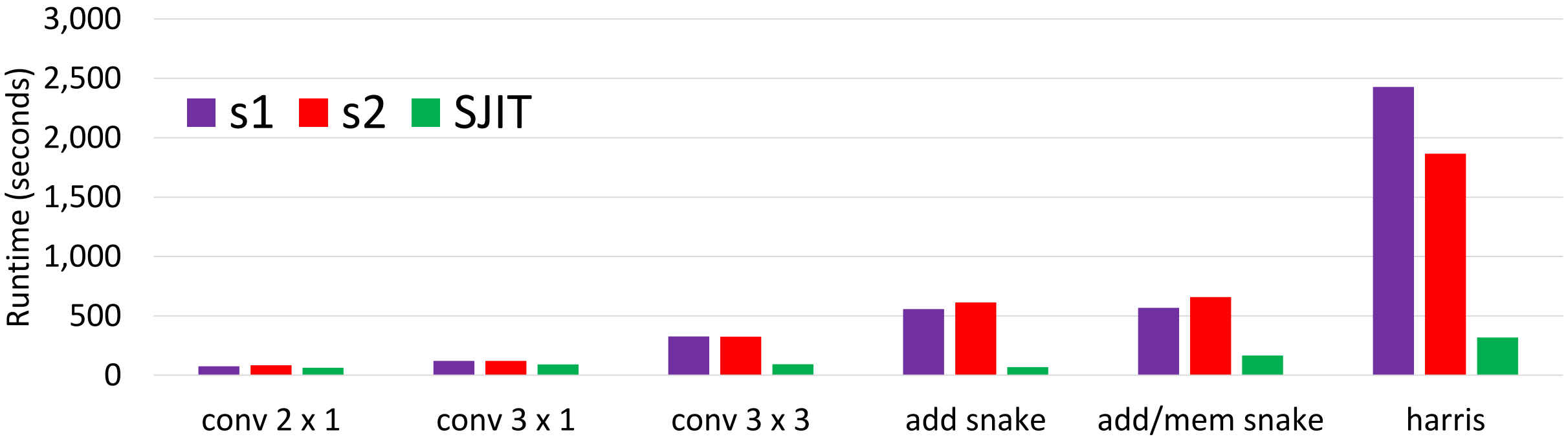
Next-Gen CGRA Generator

- See next talk

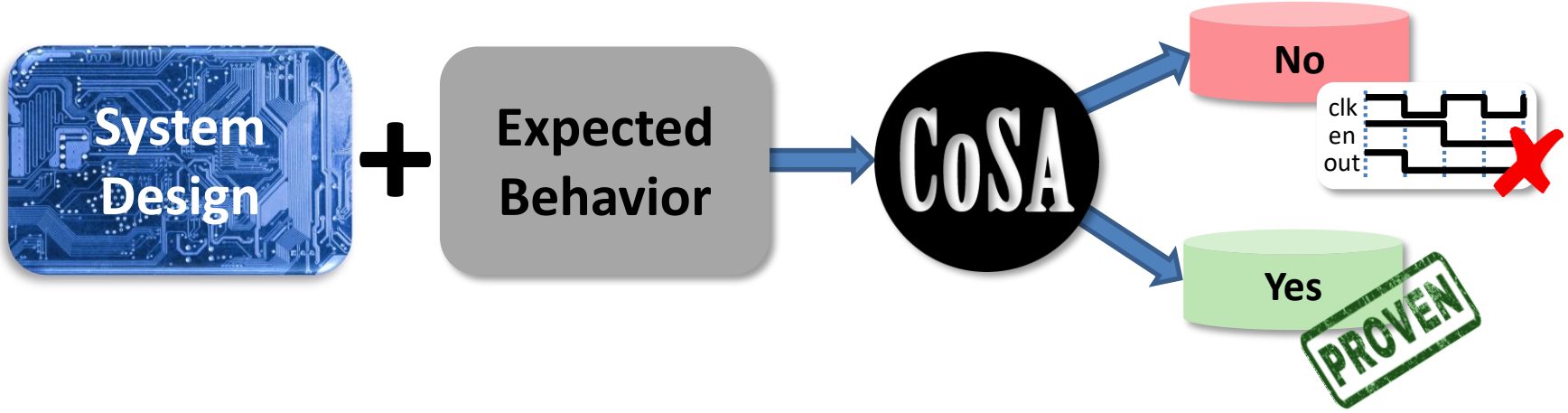


Using Just-In-Time Compilation to Accelerate CGRA Simulations

- Problem: Statically programmable datapaths are slow in conventional hardware simulators
- Solution: just in time (JIT) compilation



CoSA: CoreIR Symbolic Analyzer



CoSA is an SMT-based symbolic model checker for hardware design

- Open-source, and written in Python
- Supports multiple input formats including: **CoreIR**, **Verilog**, and **SystemVerilog**
- Provide state-of-the-art formal analyses, including: **Linear Temporal Logic** (LTL) and **invariant** verification, **equivalence checking**, proving capabilities, fault analysis, and more…

Testing and Validation

- See last talk

Fault: A Python Framework for Agile Hardware Verification using Executable Specifications, Formal Methods, and Metaprogramming

Lenny Truong - lenny@stanford.edu
StanfordAHA Intel Annual Review 10/15/2018

We Have a Complete System, But the Journey is Just Beginning

- Thanks for helping with this project

- It is great to be part of the next revolution in VLSI design