Synthesizing Programmable Accelerators: A Full-Stack Perspective

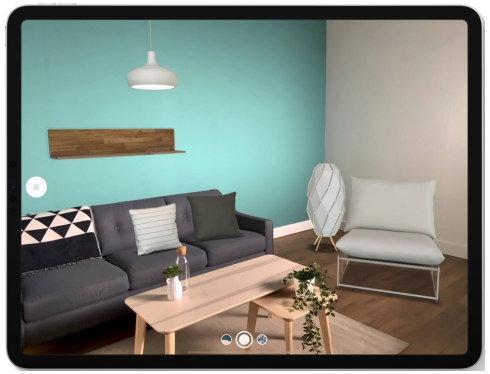
Jian Weng, Tony Nowatzki

PolyArch Research Group

University of California, Los Angeles

Nov. 5th, 2022

Increasing Demands on Compute Power



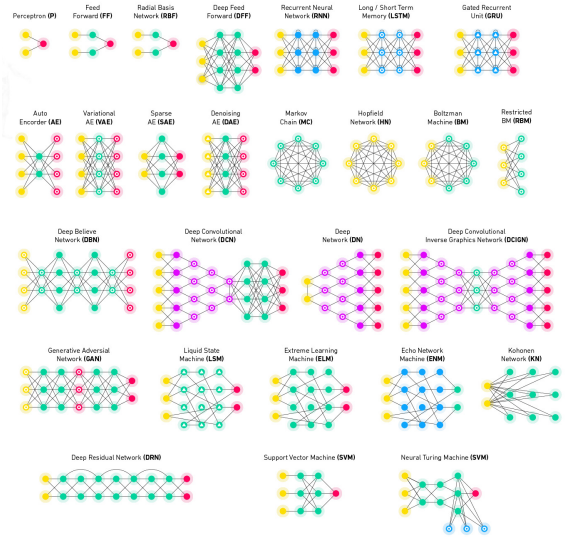
- Available on App Store.
- Virtually place 3D models in your own space.

IKEA Place

ARCore

- AR/VR Unity SDK.
- Compatible with Xiaomi.

Google ARCore



- The sizes of models keep growing
- Lengthy training time and high inference latency

ML Models

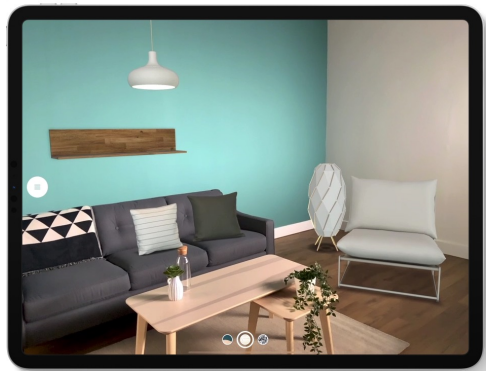
- Real-time Ray-Tracing
- 2K->4K->8K

High Resolution Gaming Rendering

- Emerging domain
- Intensive computation on big integers

Zero Knowledge Proof Protocols

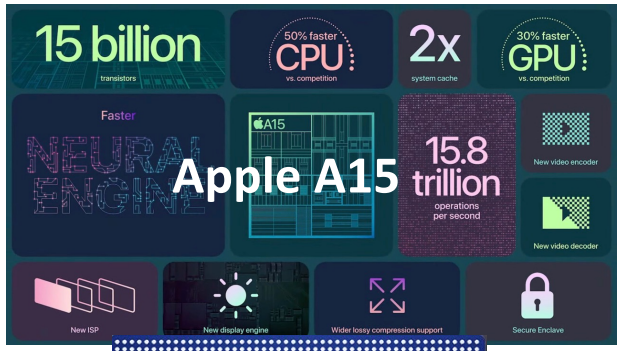
Specialized accelerators are the answer!



IKEA Place

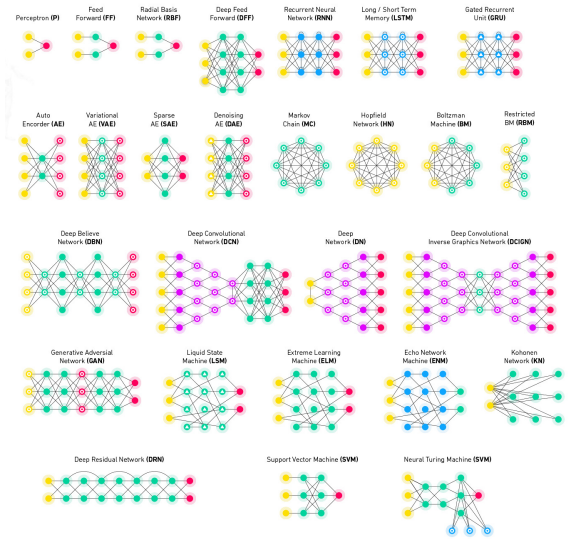
ARCore

Google ARCore

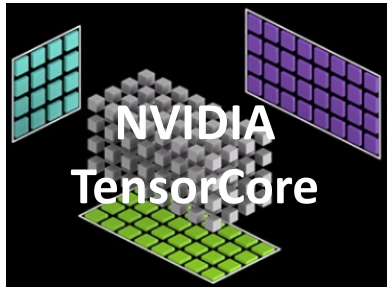


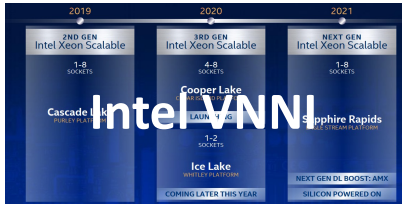
Apple A15

QualComm SnapDragon 888

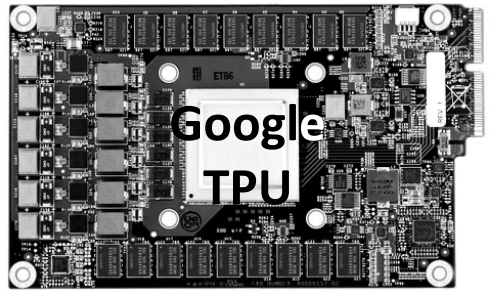


ML Models



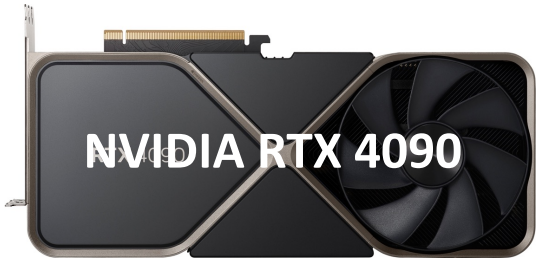


Intel VNNI

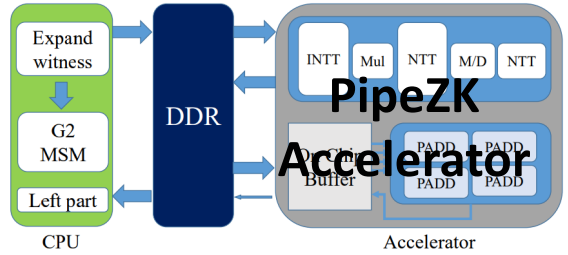


Google TPU

High Resolution Gaming Rendering

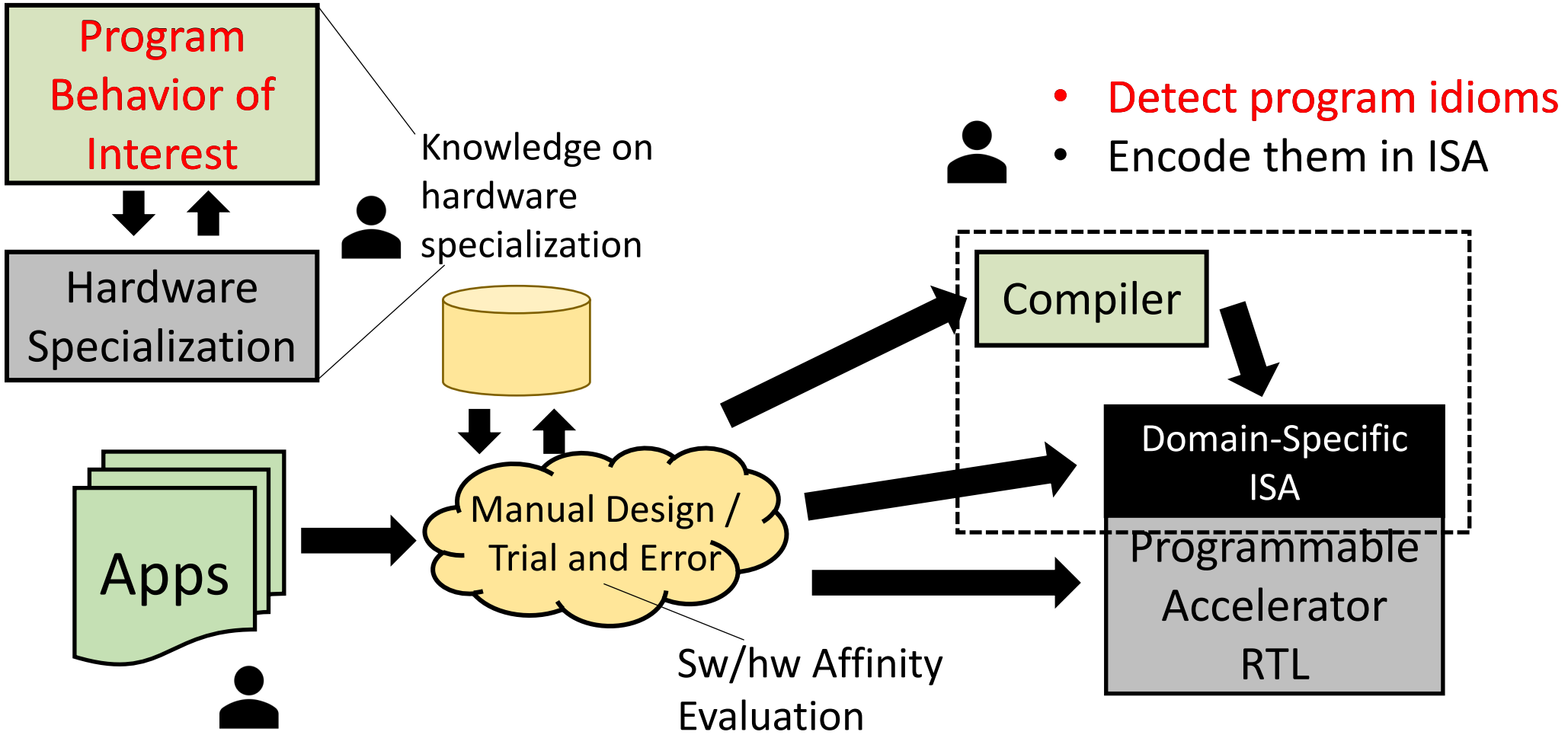


Zero Knowledge Proof Protocols



PipeZK Accelerator

Domain-Specific Accelerator Design



Insights and Principles

- Idiomatic Specialization
  - Application domains **do not** define an accelerator.

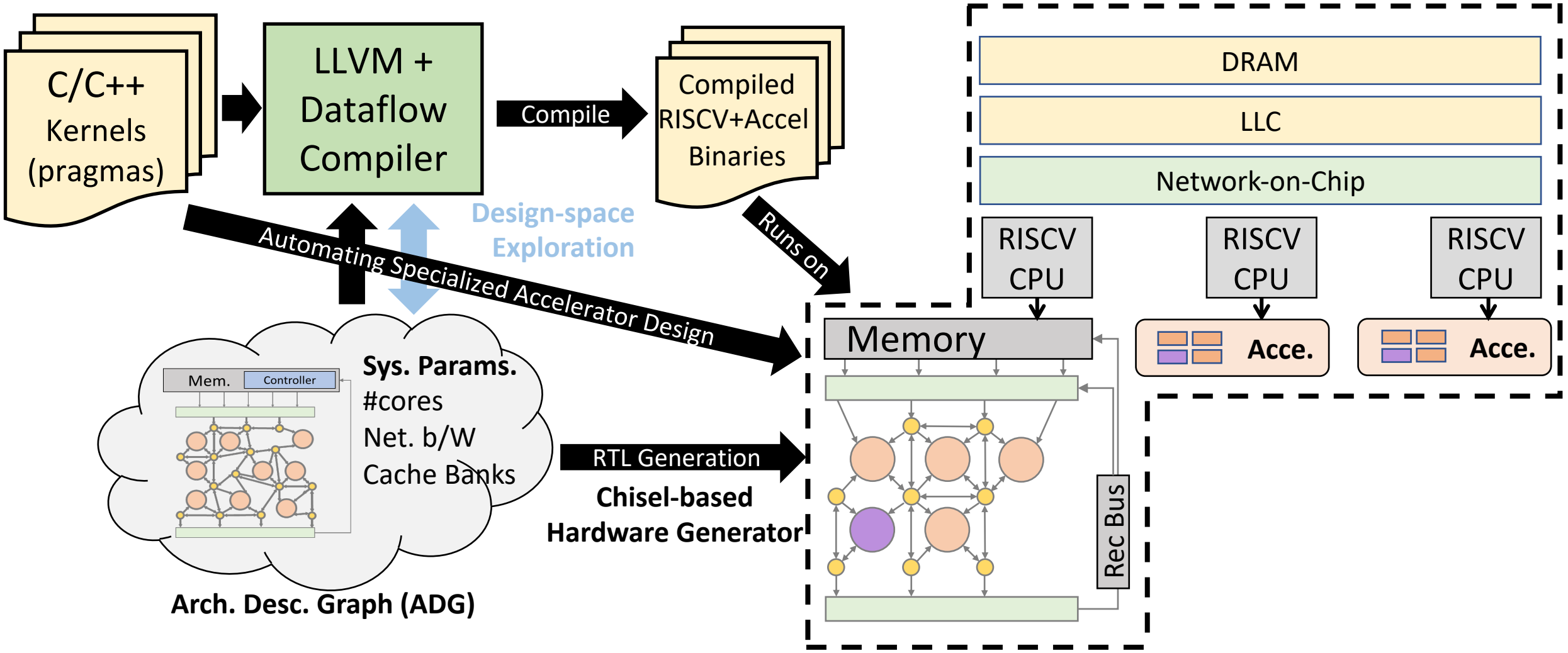
  👉 Computing behaviors of interests (computing idioms) **do**.
- Modularity (Define the design space)
  - Each idiom-specialized feature should be **modular**.

  👉 Hardware and compilation should be modularly flexible.
- Generality and Performance
  - Avoid Von-Neumann overheads while retaining generality.

  👉 **Decoupled-Spatial** Paradigm

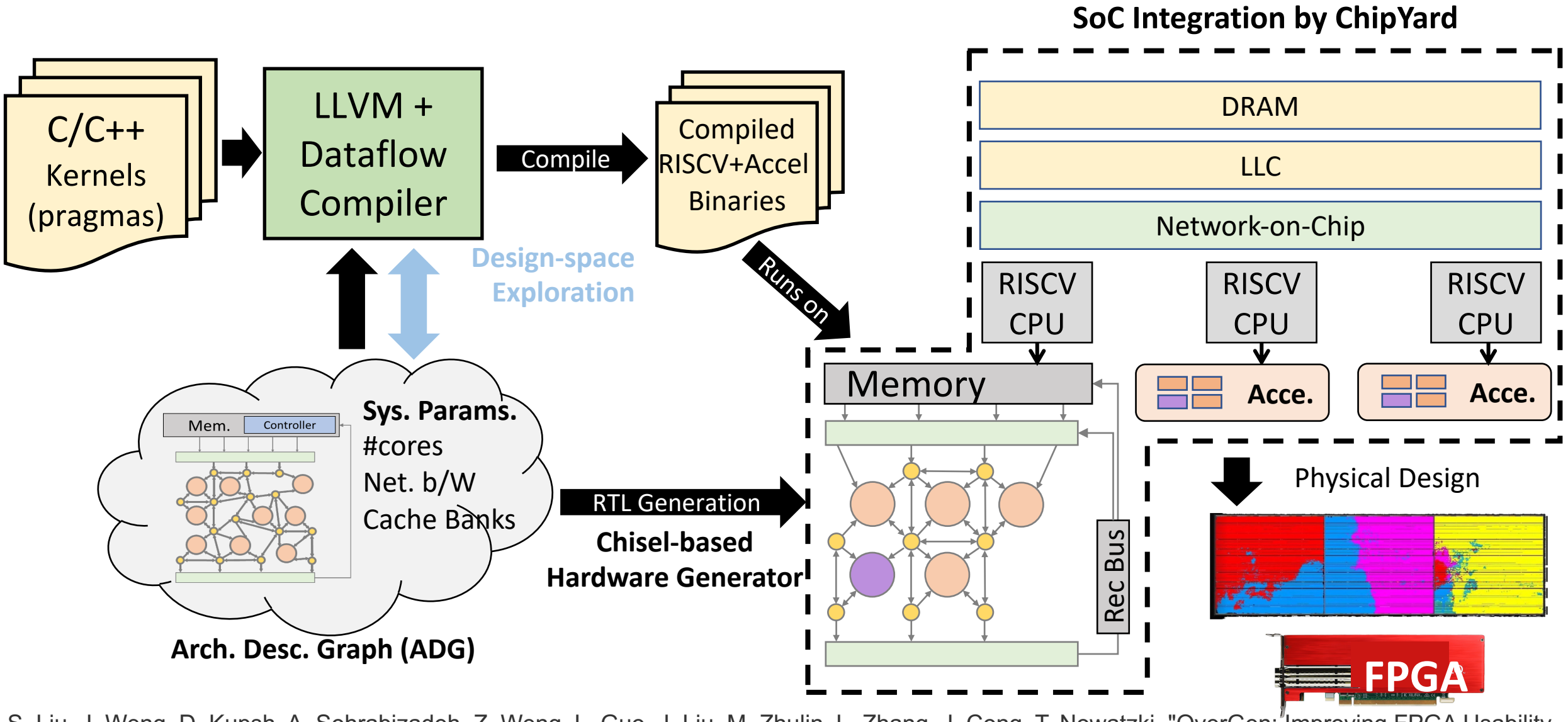
Decoupled-Spatial Architecture Design Automation

Domain-Specific



J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah and T. Nowatzki, "DSAGEN: Synthesizing Programmable Spatial Accelerators", 47th ISCA, 2020, pp. 268-281, doi: 10.1109/ISCA45697.2020.00032. (IEEE Top Picks Honorable Mentions)

Specialized Overlay Generation for FPGA Programming



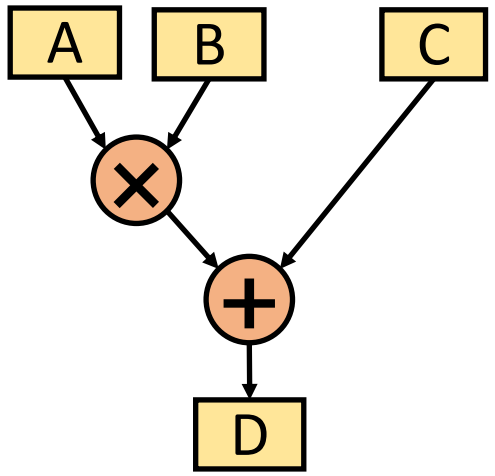
S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Weng, L. Guo, J. Liu, M. Zhulin, L. Zhang, J. Cong, T. Nowatzki, "OverGen: Improving FPGA Usability through Domain-specific Overlay Generation", 2022 55th MICRO, pp. 35-56. doi: 10.1109/MICRO56248.2022.00018 (Best Paper Runner-up)

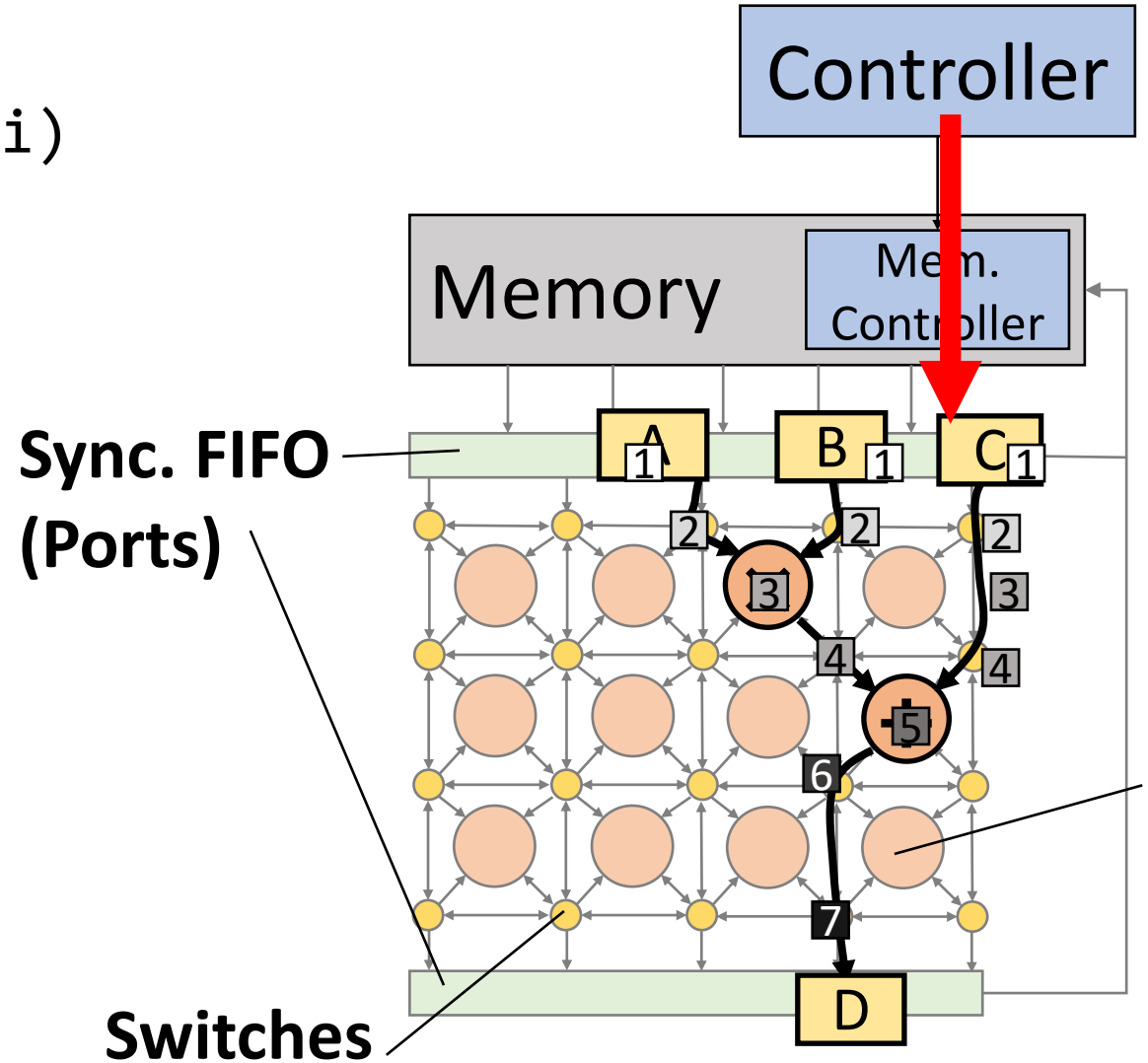
Roadmap

- Motivation
- **Background**
  - **Decoupled-Spatial Execution**
  - **Design Space**
- Compiler
- Design Space Exploration
- Evaluation
- Future Work

Decoupled-Spatial Paradigm: Spatial Architecture

```
for (int i = 0; i < n; ++i)
    c[i] += a[i] * b[i];
```





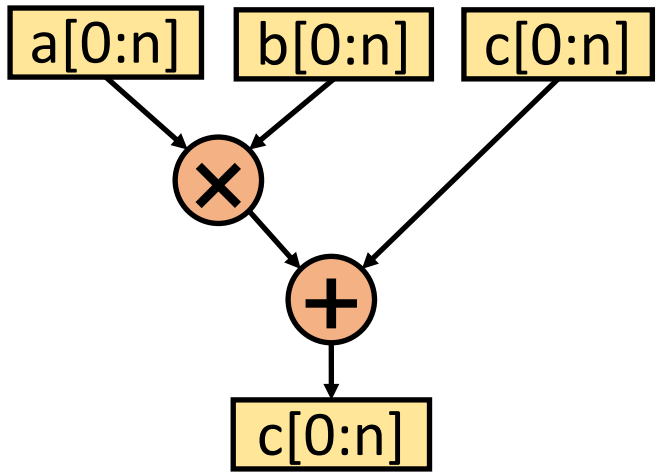
Sync. FIFO (Ports)

Processing Elements

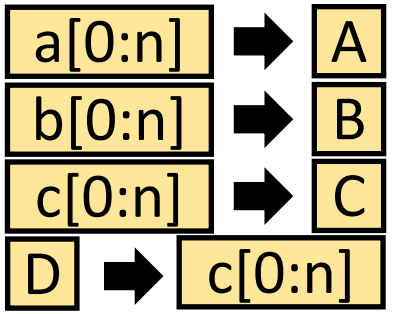
Switches

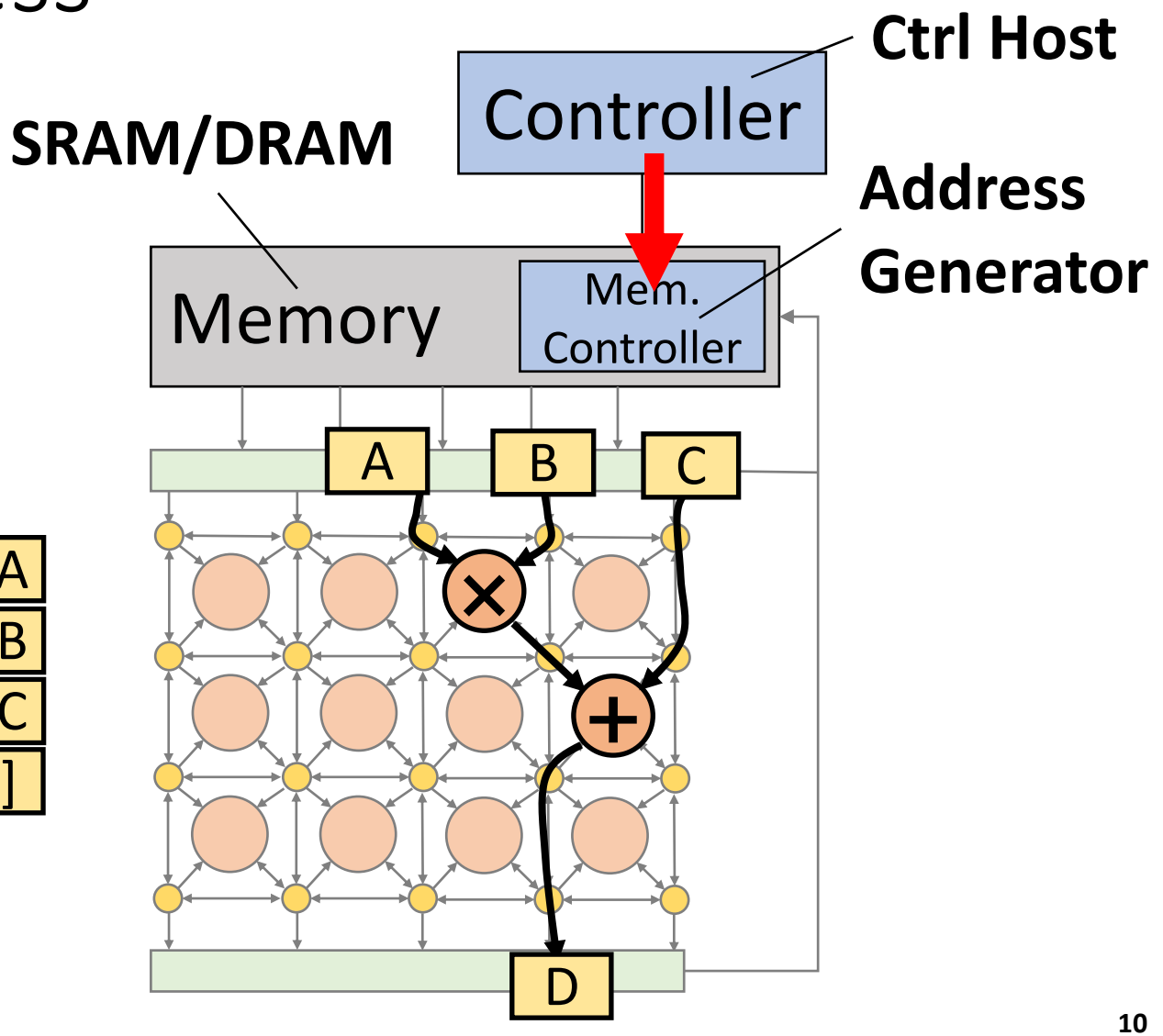
Decoupled-Spatial Paradigm: Decoupled Memory Access

```
for (int i = 0; i < n; ++i)
    c[i] += a[i] * b[i];
```



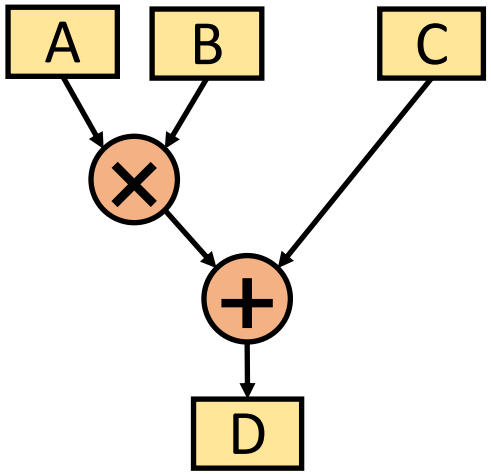
Mem. Access:

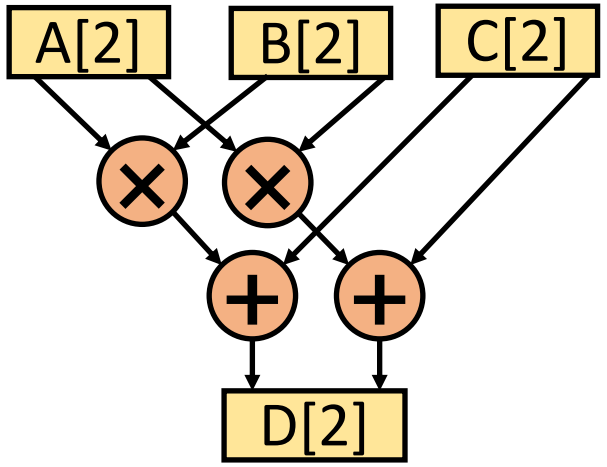


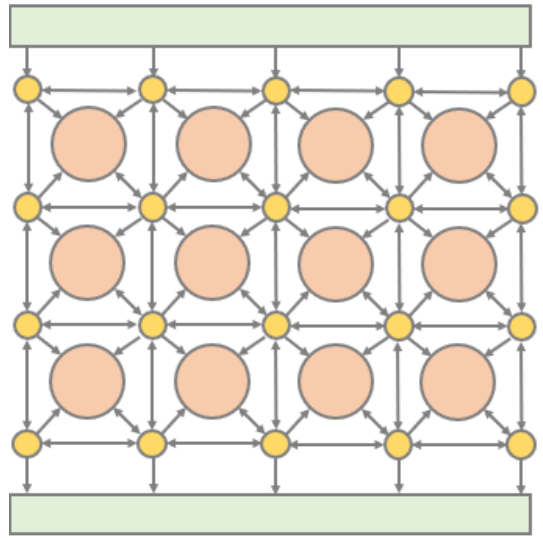


“Vectorization” on Spatial Architecture

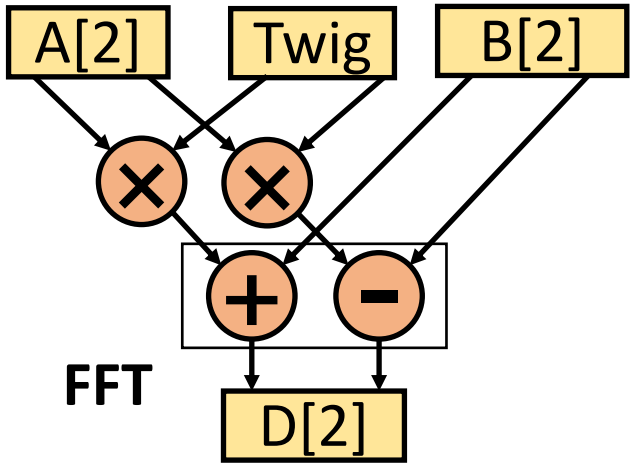
- Duplicate resource (Comp. & Mem B/w) occupied to vectorize

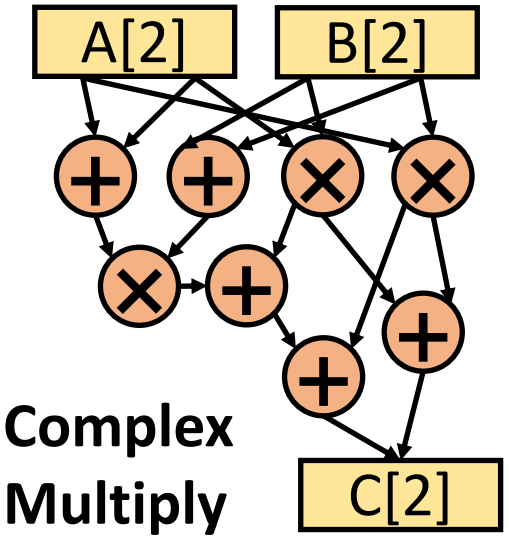


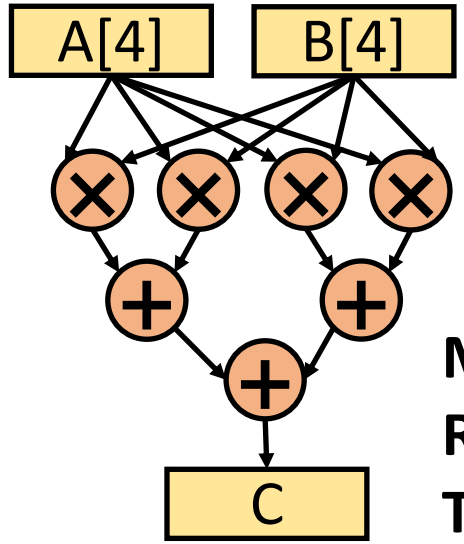




- Even more flexibility







Address Generator: Coarser Grain Memory Access

- ~~Scalar? Vector?~~
- “Stream”: A set of memory accesses under loop nest
- Dense Memory Generator

```
for (i=0; i<n; ++i)
  for (j=0; j<m; ++j)
    for (k=0; k<p; ++k)
      // request a[i*si+j*sj+k*sk]
```

- Sparse

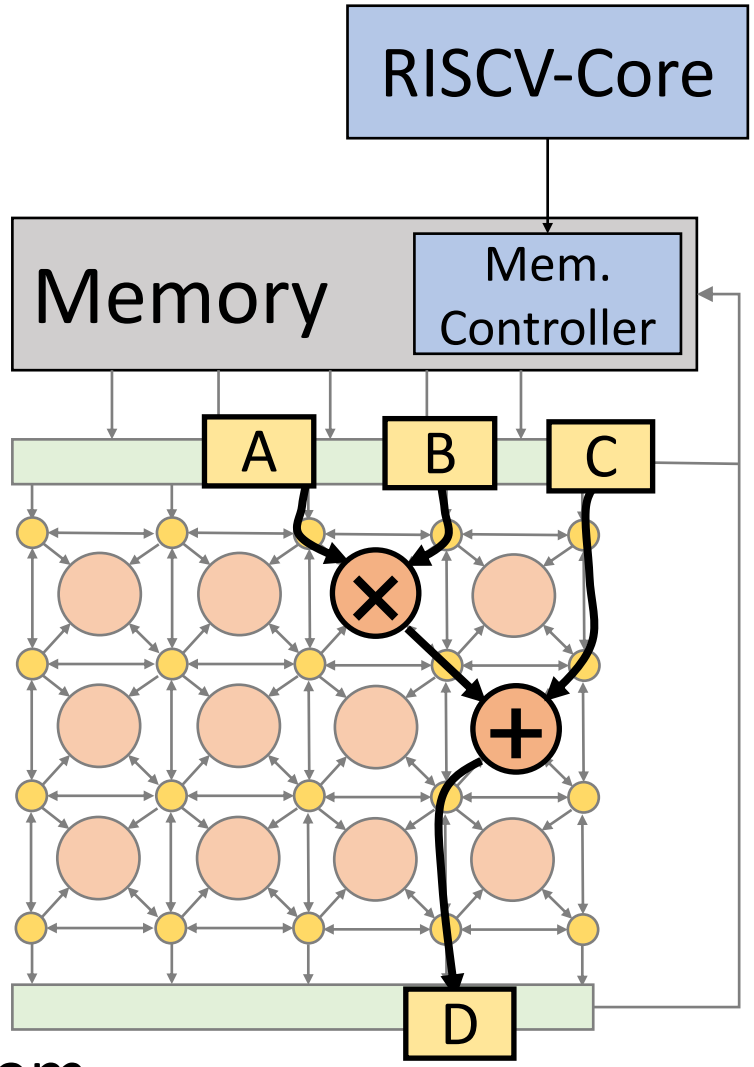
```
for (i=0; i<m; ++i)
  for (j=0; j<n[i]; ++j)
    // request a[b[i]+j]
```

Decoupled-Spatial Paradigm: Summary

```
for (int i = 0; i < n; ++i)
    c[i] += a[i] * b[i];
```

```
main:
  li $rs3, 0
loop:
  load $rs1, a[i]
  load $rs2, b[i]
  mul $rs1, $rs1, $rs2
  load $rs0, c[i]
  add $rs0, $rs1
  store $rs1, c[i]
  add $rs3, $rs3, 1
  blt $rs3, n, loop
```

```
main:
  ...
  load config
  a[0:n] -> A
  b[0:n] -> B
  c[0:n] -> C
  D -> c[0:n]
  barrier
```



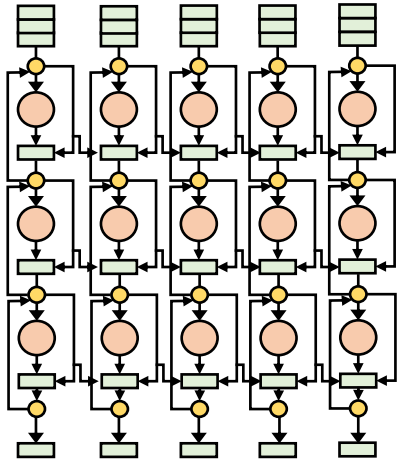
\+ Simpler PE, better memory b/w utilization

\- Different execution model for comp. and mem.

Topology: Architecture Description Graphs

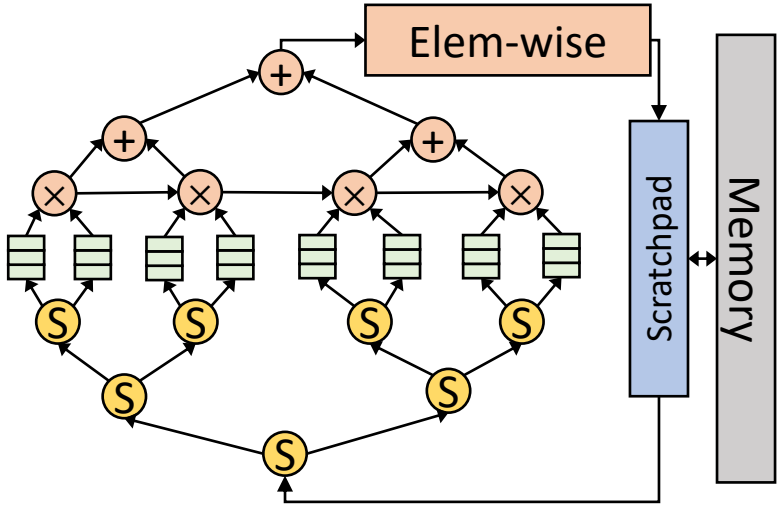
General Purpose:

SIMD/Vector-Style



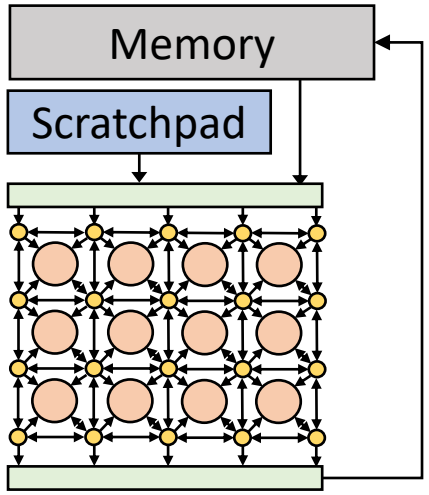
Plasticine-PCU
ISCA'17

ReduceTree-Style



MAERI
ASPLOS'18

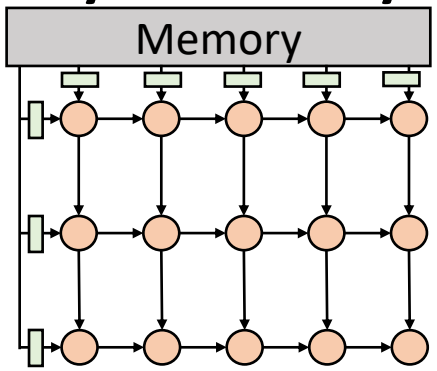
Mesh-Style



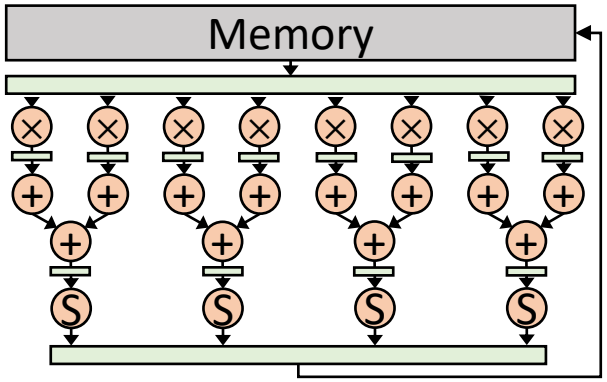
Softbrain
ISCA'17

App or Domain Specific:

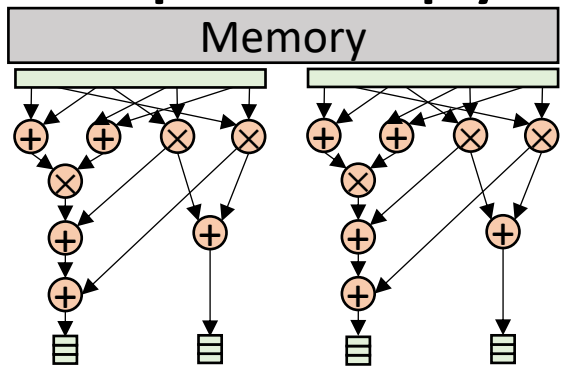
Systolic Array



Matrix-Vector



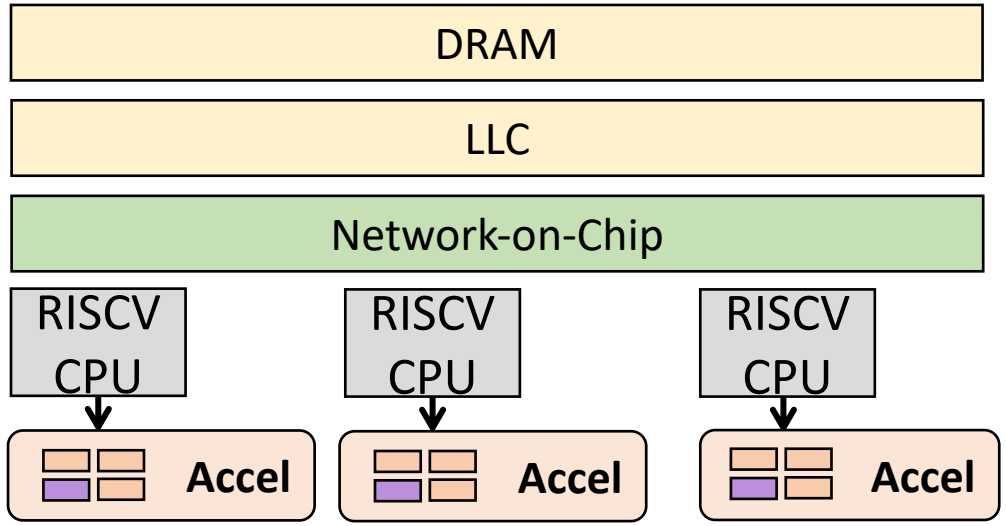
Complex Multiply

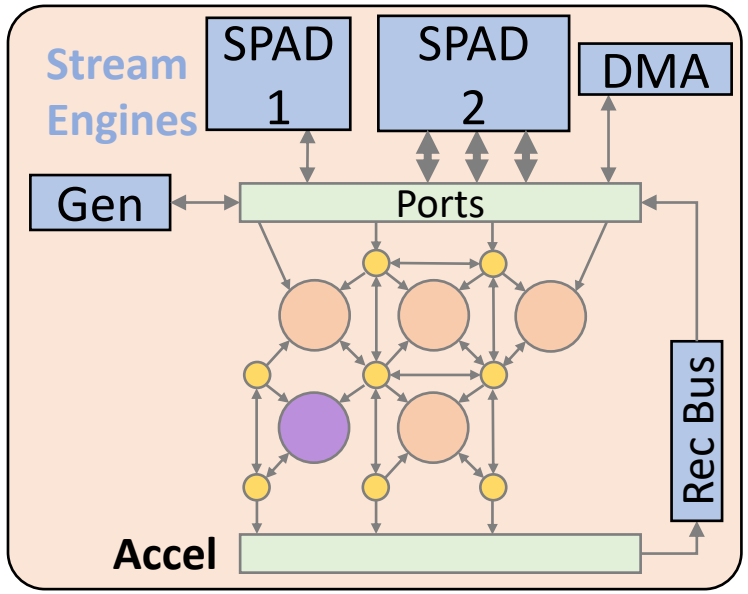


Hardware Primitives

- Process Elem.
- Memory
- Switch
- Scratchpad
- Sync. Elem.

Parameters: System & Accelerator





- System Params
  - L2 Banks
  - L2 Capacity
  - NoC Bandwidth
  - # Cores

Key Tradeoffs:
Few Big vs More Small Cores?
L2 Size vs Scratchpad Size?
More Compute vs More Network?
Parallelism vs Generality?

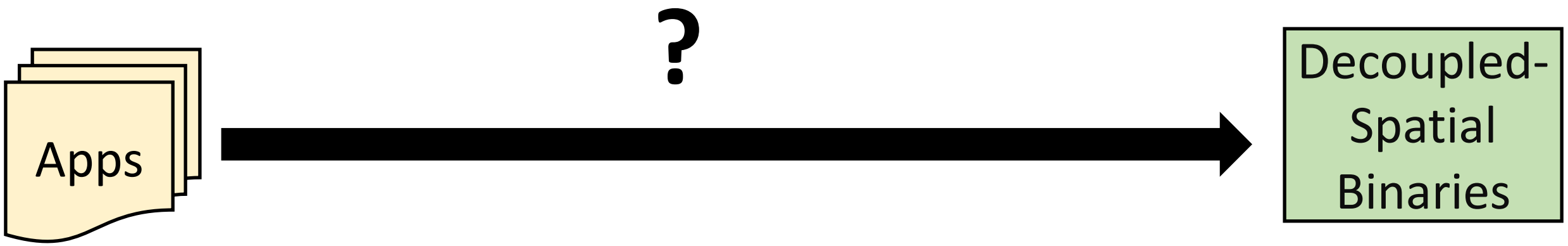
How? Compiler/DSE must reason about model memory and compute bandwidths + data reuse

- Spatial Params
  - Common (PEs/switches/mem/ports)
    - Width, Topolgy
  - Proc Elements (PEs)
    - Func-unit Set, Static/dynamic Sched., Dedicated/Shared
  - Memory Units (stream engines)
    - DMA, SPADs, Recurrence Bus, Generate
    - Capacity
    - Address Patterns Supported (affine, indirect)

Roadmap

- Motivation
- Background
- **Compiler**
  - **High-Level Abstraction**
  - **Idiom-Oriented Transformations**
- Design Space Exploration
- Evaluation
- Future Work

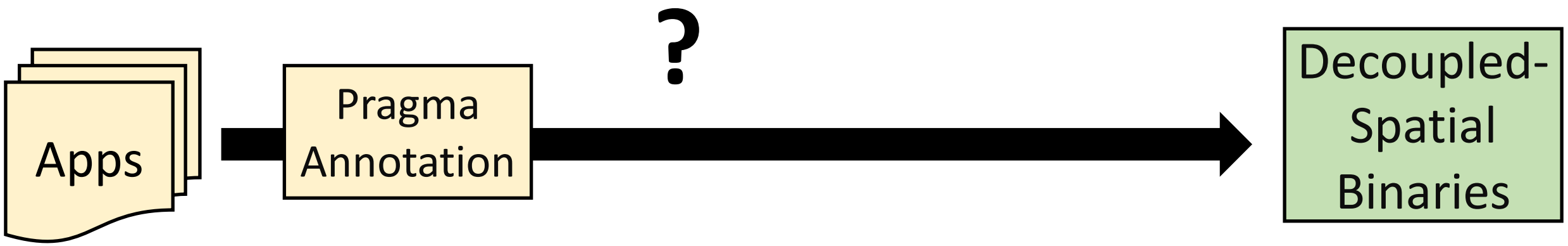
Compiling High-Level Lang. to Decoupled Spatial



Goal: Compiling high-level language to *decoupled-spatial* architectures.

- Decoupled-Spatial
  - Program behaviors are specialized (executed) on different components.
- Architectures
  - Hardware features may be optional/unavailable.

Decoupled Spatial Compilation (CONT'D)



Our Solution: Programmer Hints + Idiomatic Transformation.

- Programmer Hints
  - How the code should be decoupled?
    - Memory
    - Spatial
  - How to deal with the code concurrency?

An example of pragma annotation

```
#pragma config
{
  #pragma stream
  #pragma offload
  for (i=0; i<n; ++i) {
    c[i] += a[i] * b[i];
  }
}
```

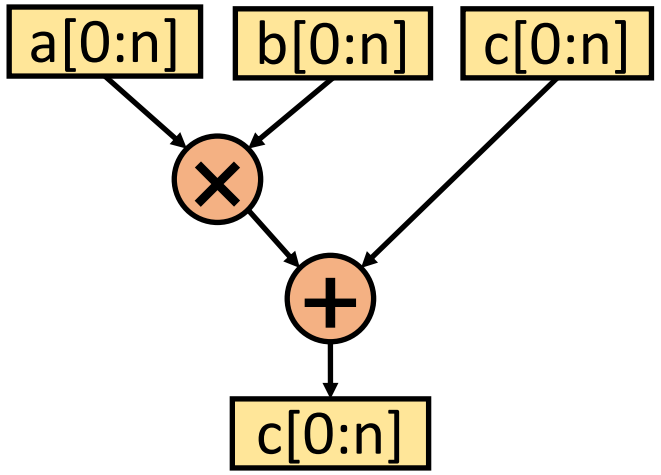
← The offloaded region in this compound body are concurrent

← The memory accesses below will be restricted

← The computational instructions below will be offloaded

Decoupling Comp. & Mem.

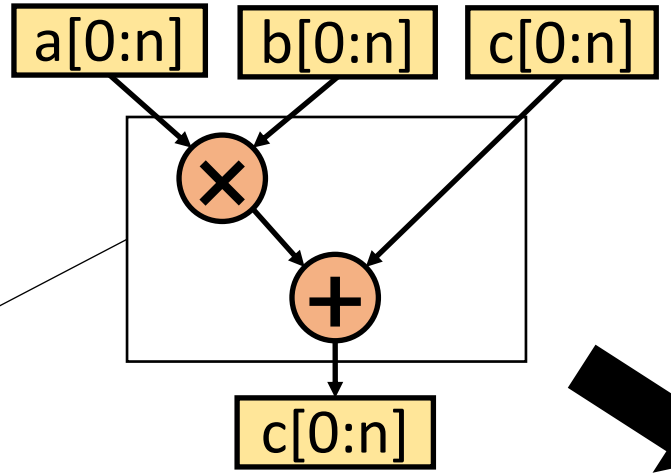
```
#pragma config
{
  #pragma stream
  #pragma offload
  for (i=0; i<n; ++i) {
    c[i] += a[i] * b[i];
  }
}
```

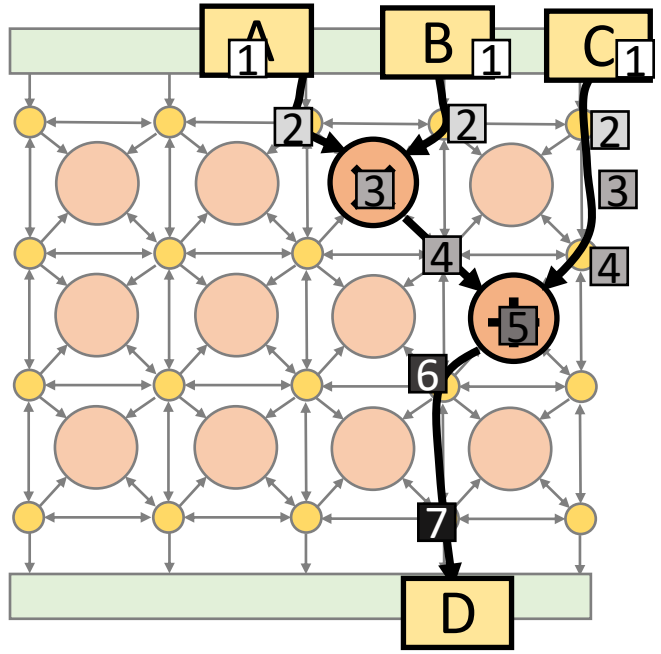



```
  %1 = config.begin()
loopi:
  %i = phi [0, preheader],
           [i.inc]
  ptr.a = getelem %a, %i
  val.a = load ptr.a
  ptr.b = getelem %b, %i
  val.b = load ptr.b
  ptr.c = getelem %c, %i
  val.c = load %c
  %2 = mul val.a, val.b
  %3 = add %2, val.c
  store ptr.c, %3
  br i<n, loopi, !offload, !stream
loop.epilog:
  config.end(%1)
```

Analysis & Transformation

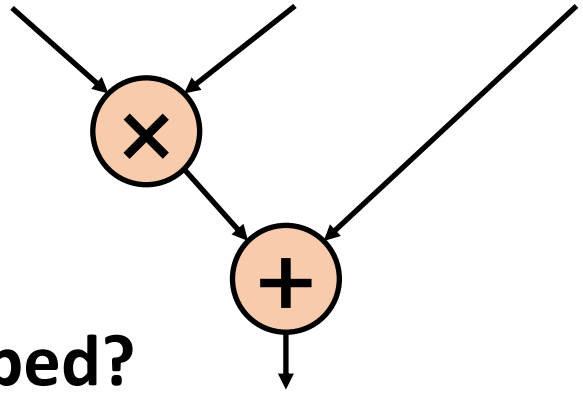
```
   %1 = config.begin()
loopi:
   %i = phi [0, preheader],
            [i.inc]
   ptr.a = getelem %a, %i
   val.a = load ptr.a
   ptr.b = getelem %b, %i
   val.b = load ptr.b
   ptr.c = getelem %c, %i
   val.c = load %c
   %2 = mul val.a, val.b
   %3 = add %2, val.c
   store ptr.c, %3
   br i<n, loopi, !offload, !stream
loop.epilog:
   config.end(%1)
```

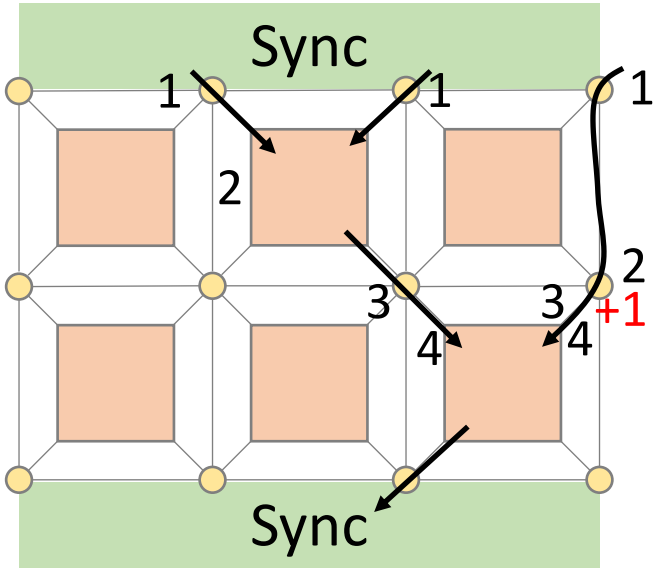




Spatial Mapper

Spatial Mapping





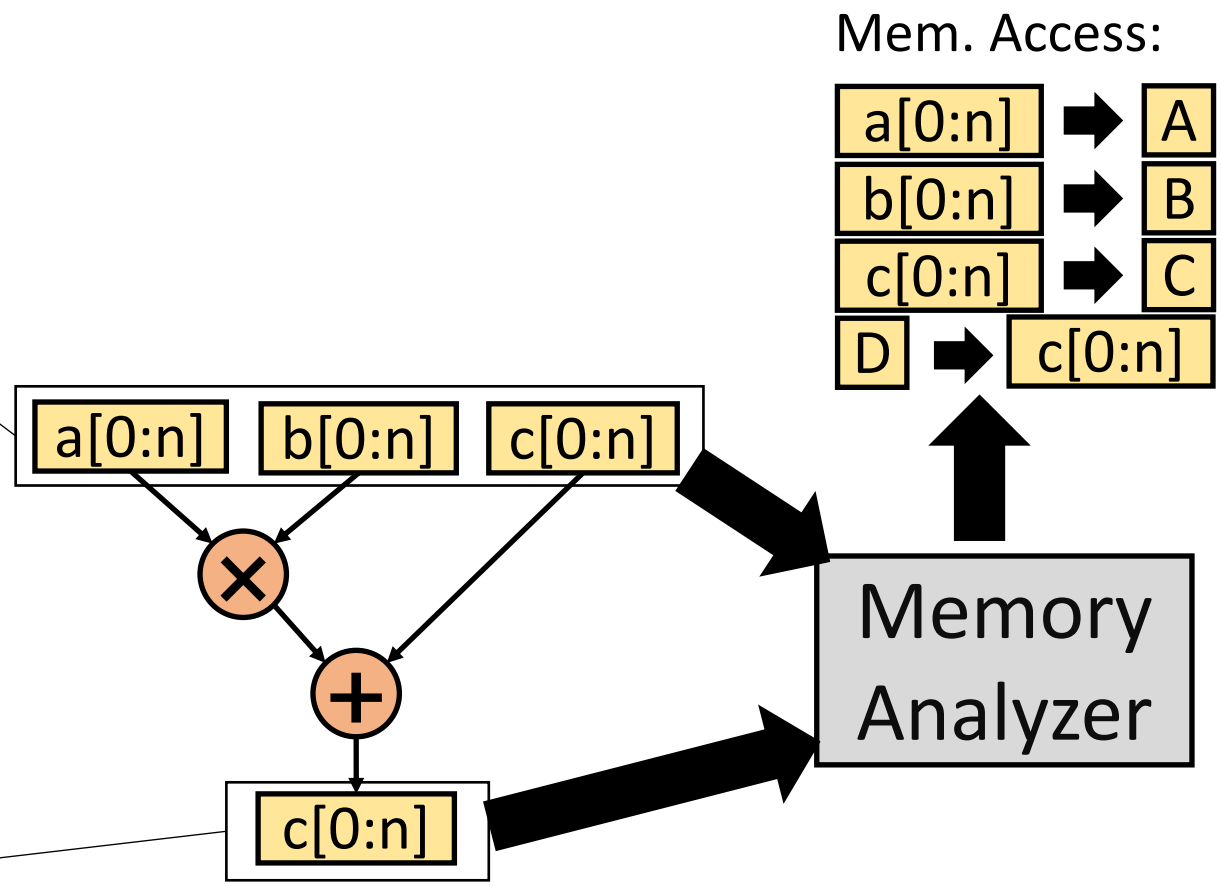
How is the dependence graph of computational instructions mapped?

1. Placement: Map instruction to PE's with corresponding capability.
2. Routing: Routing the dependence edges thru the spatial network.
3. Timing: If necessary, balance the timing of data arrival

- If one of 1-3 is not successful, revert some nodes and repeat 123
- Estimate the performance: #Inst x (Activity Ratio)
  - Execution is driven by data availability

Analysis & Transformation

```
  %1 = config.begin()
loopi:
  %i = phi [0, preheader],
           [i.inc]
  ptr.a = getelem %a, %i
  val.a = load ptr.a
  ptr.b = getelem %b, %i
  val.b = load ptr.b
  ptr.c = getelem %c, %i
  val.c = load %c
  %2 = mul val.a, val.b
  %3 = add %2, val.c
  store ptr.c, %3
  br i<n, loopi, !offload, !stream
loop.epilog:
  config.end(%1)
```



J. Weng, S. Liu, D. Kupsh and T. Nowatzki, "Unifying Spatial Accelerator Compilation With Idiomatic and Modular Transformations," in IEEE Micro, vol. 42, no. 5, pp. 59-69, 1 Sept.-Oct. 2022, doi: 10.1109/MM.2022.3189976.

Address Generator: Coarser Grain Memory Access

- ~~Scalar? Vector?~~
- "Stream": A set of memory accesses under loop nest
- Dense Memory Generator

```
for (i=0; i<n; ++i)
  for (j=0; j<m; ++j)
    for (k=0; k<p; ++k)
      // request a[i*si+j*sj+k*sk]
```

- Sparse

```
for (i=0; i<m; ++i)
  for (j=0; j<n[i]; ++j)
    // request a[b[i]+j]
```

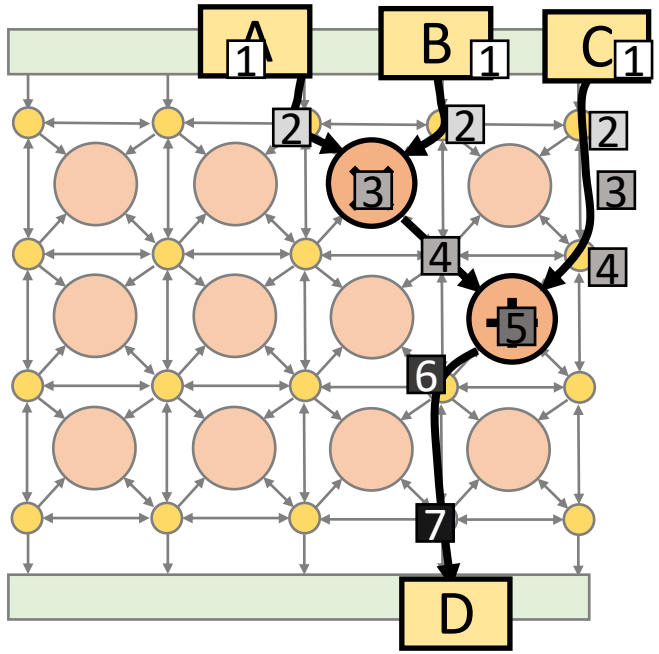
Analysis & Transformation

```
   %1 = config.begin()
loopi:
   %i = phi [0, preheader],
            [i.inc]
   ptr.a = getelem %a, %i
   val.a = load ptr.a
   ptr.b = getelem %b, %i
   val.b = load ptr.b
   ptr.c = getelem %c, %i
   val.c = load %c
   %2 = mul val.a, val.b
   %3 = add %2, val.c
   store ptr.c, %3
   br i<n, loopi, !offload, !stream
loop.epilog:
   config.end(%1)
```

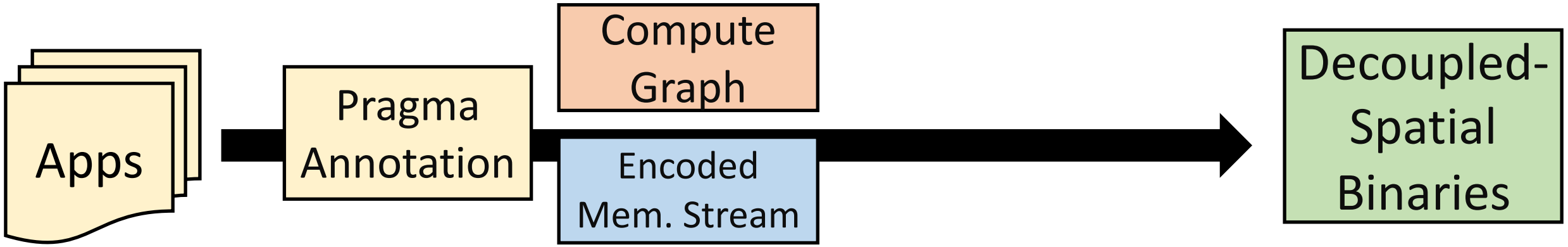
```
Load Config
a[0:n] -> A
b[0:n] -> B
c[0:n] -> C
D -> c[0:n]
Barrier
```

Mem. Access:

a[0:n]	➡	A
b[0:n]	➡	B
c[0:n]	➡	C
D	➡	c[0:n]



Decoupled Spatial Compilation (CONT'D)



Our Solution: Programmer Hints + Idiomatic Transformation.

- Programmer Hints
- Idiomatic (&Modular) Transformation
  - Analyze the software idiom
  - Inspect the hardware capability
  - Rewrite it accordingly
- Specialized xform
- Fallback xform

Modular Transformation & Fallback

```
// sparse memory access
for (i=0; i<n; ++i)
  // request a[b[i]]
```

- Escape using the sparse unit when not available.

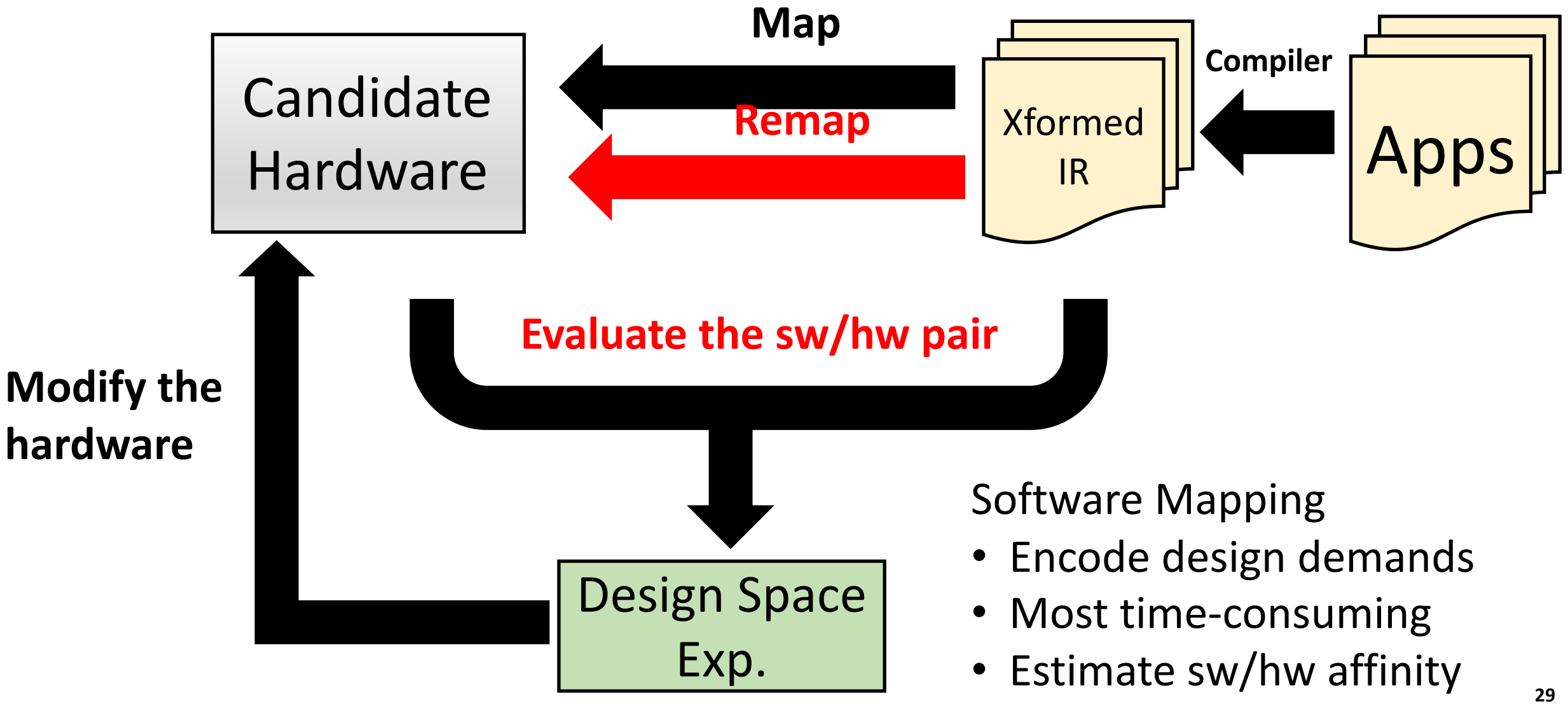
```
// specialized xform
linear(/*arr=*/b,
       /*len=*/n,
       /*dst=*/IndPort)
sparse(/*idx=*/IndPort,
       /*arr=*/a,
       /*len=*/n,
       /*dst=*/DataPort)
```

```
// fallback xform
for (i=0; i<n; ++i)
  scalar(/*data=*/a[b[i]],
         /*dest=*/DataPort);
```

Roadmap

- Motivation
- Background
- Compiler
- **Design Space Exploration**
  - **Overview**
  - **Synthesizing Memory Hierarchy**
  - **Hardware Evolution**
  - **Scheduling Acceleration**
  - **Objective Evaluation**
- Evaluation
- Future Work

Automated Design Space Exploration

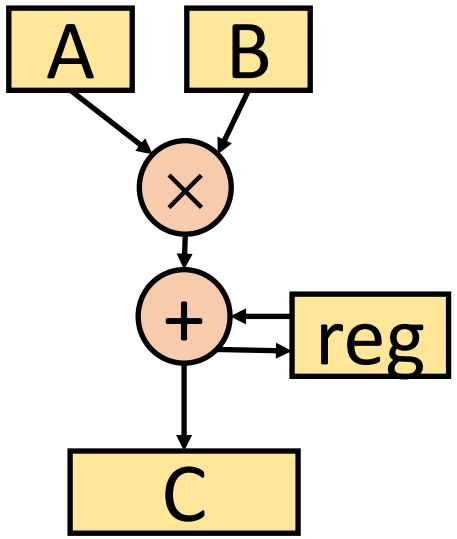


Software Mapping

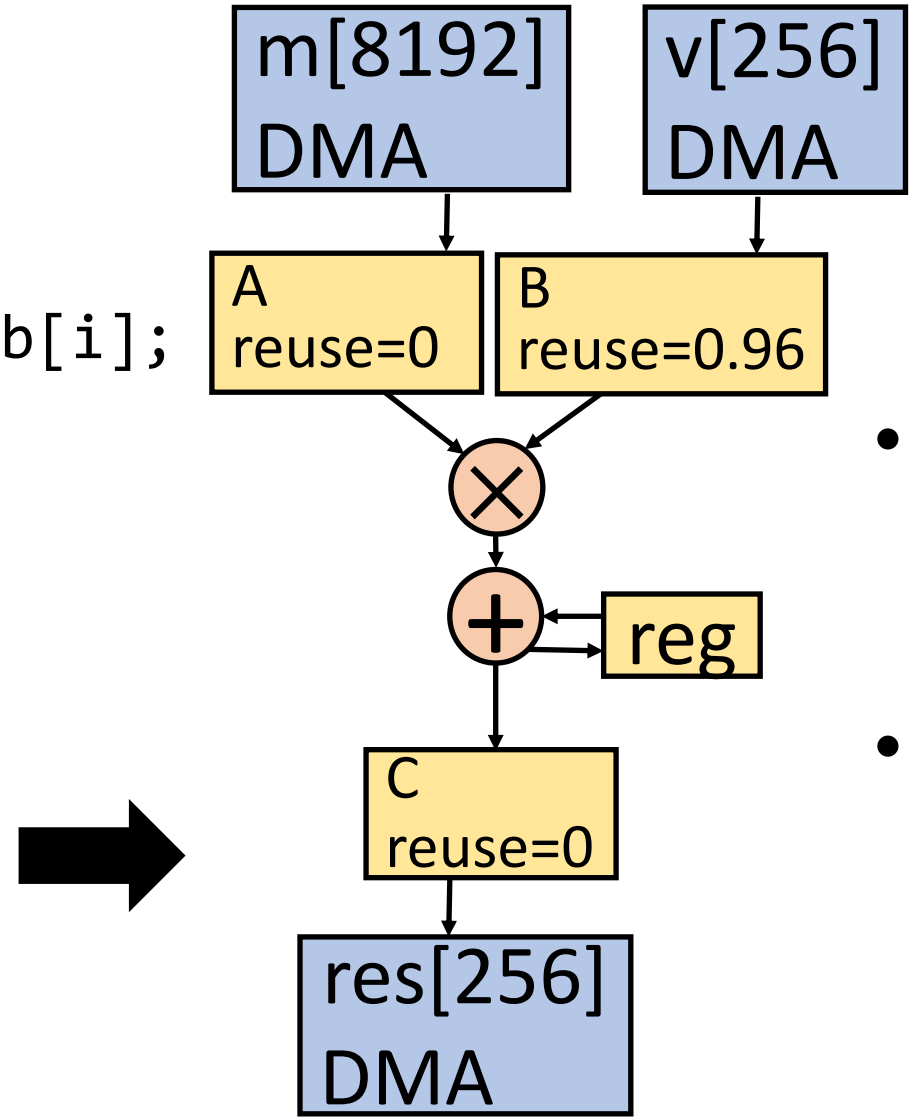
- Encode design demands
- Most time-consuming
- Estimate sw/hw affinity

Synthesizing Memory Hierarchy: Memory-Improved DFG

```
for (i=0; i<32; ++i) {
  acc=0;
  for (j=0; j<32; ++j)
    acc += a[i*n+j] * b[i];
  c[i] = acc;
}
```



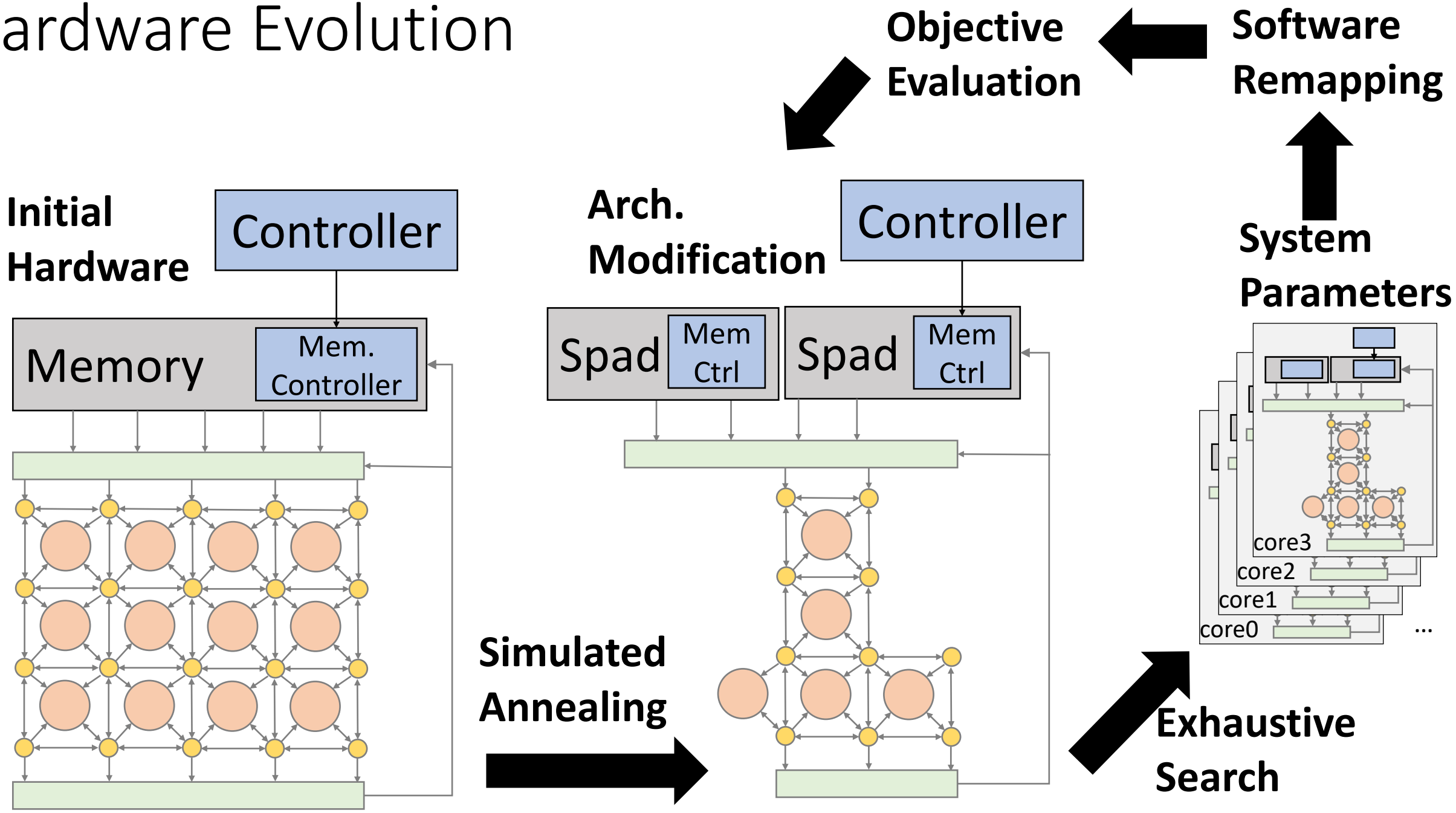
(a) Original DFG



(b) mDFG

- Additional Info
  - Array Size
  - Access Reuse
- H/w Specialization
  - Port/Mem Topology
  - L2 Bandwidth
  - SPAD Instances

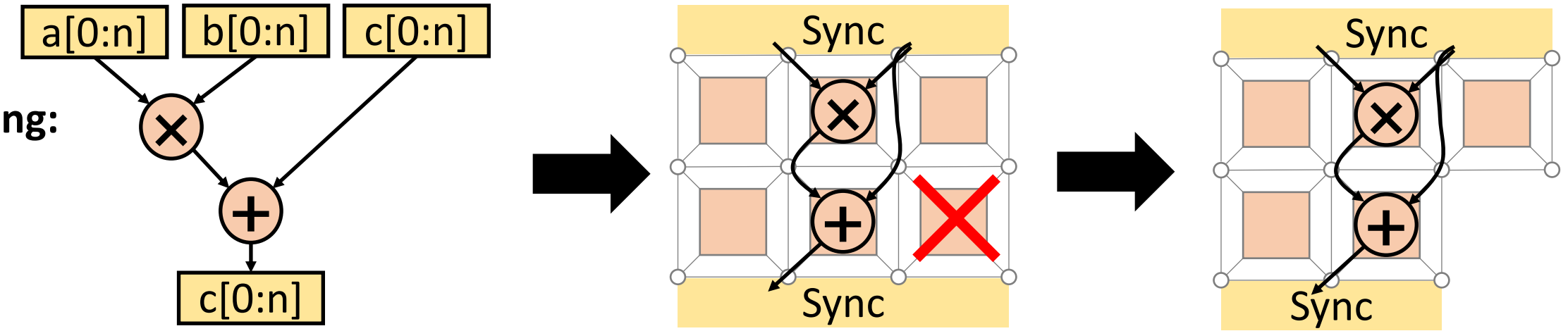
Hardware Evolution



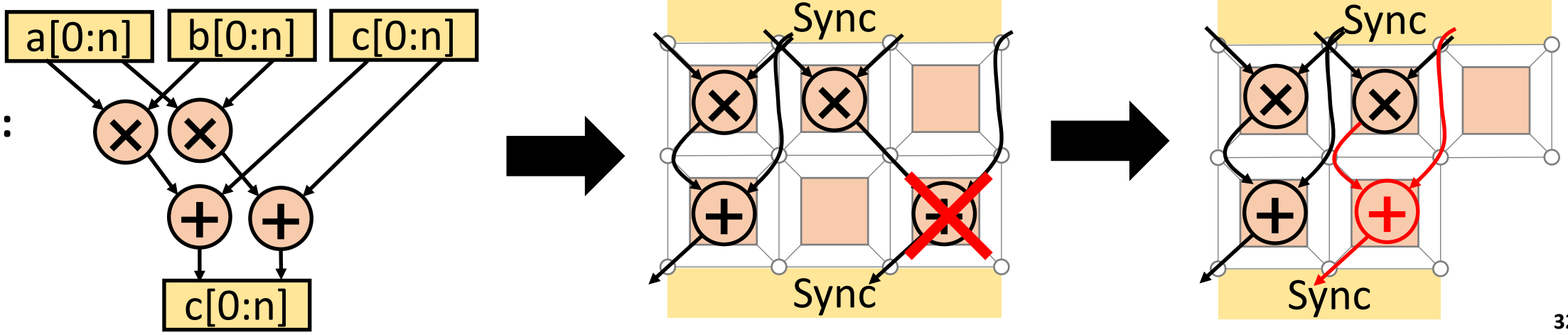
Accelerating Spatial Mapping: Reparation

```
// Original Code
for (i=0; i<n; ++i)
  c[i]+=a[i]*b[i];
```

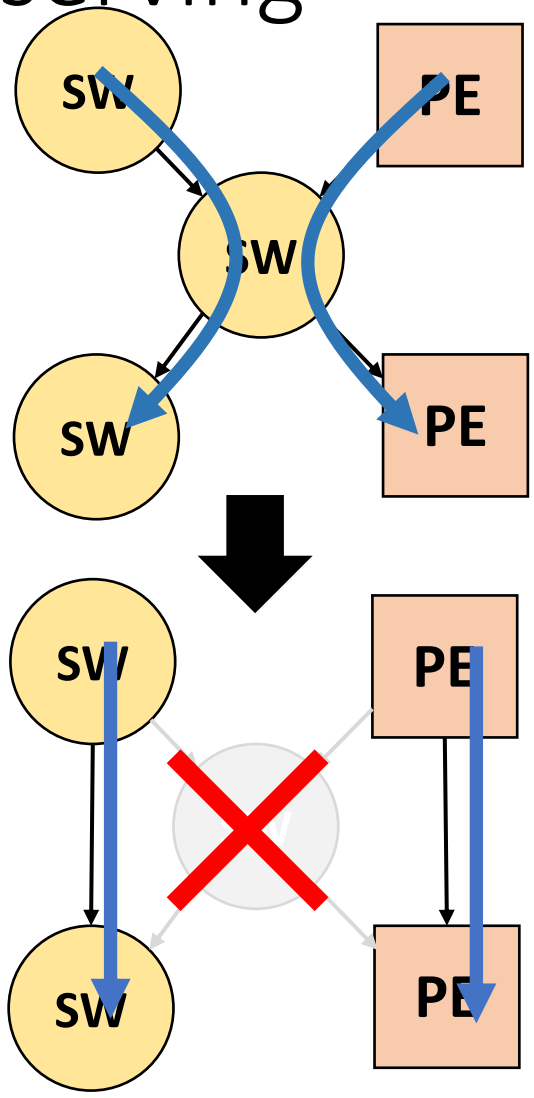
No Unrolling:



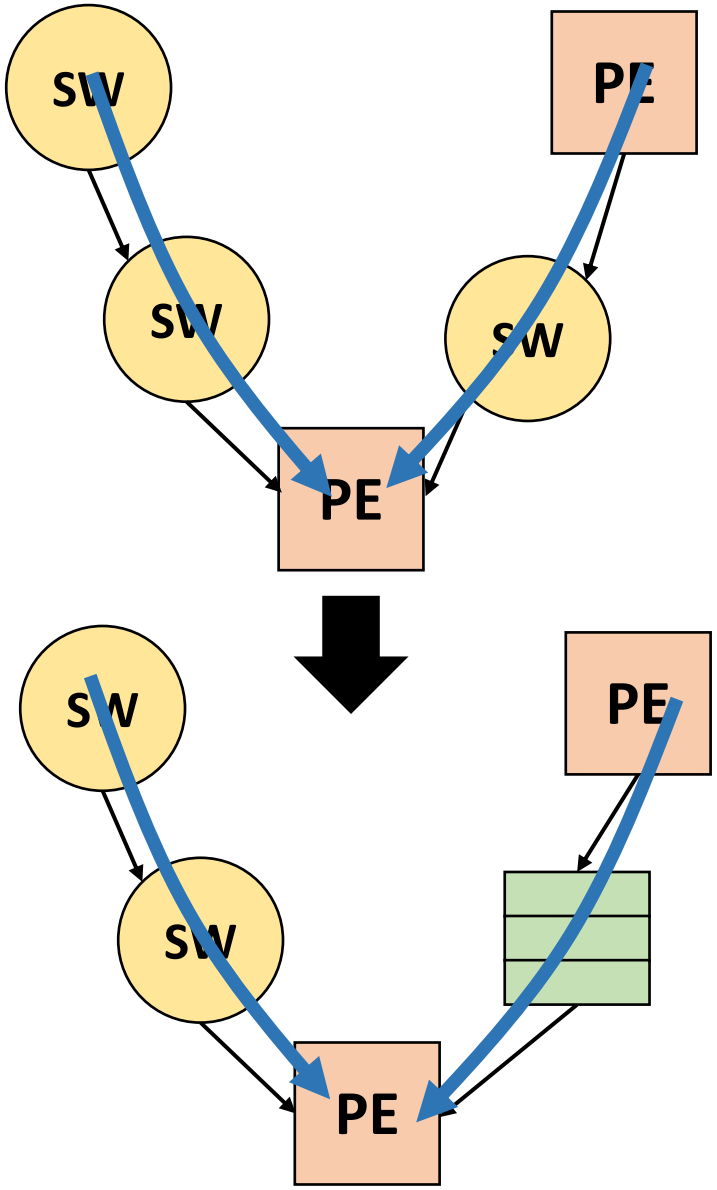
Unroll by 2:



Accelerating Spatial Mapping: Preserving



- Replace a s/w by a direct edge.



- Replace a s/w by delay FIFO to match timing.

Objective Function

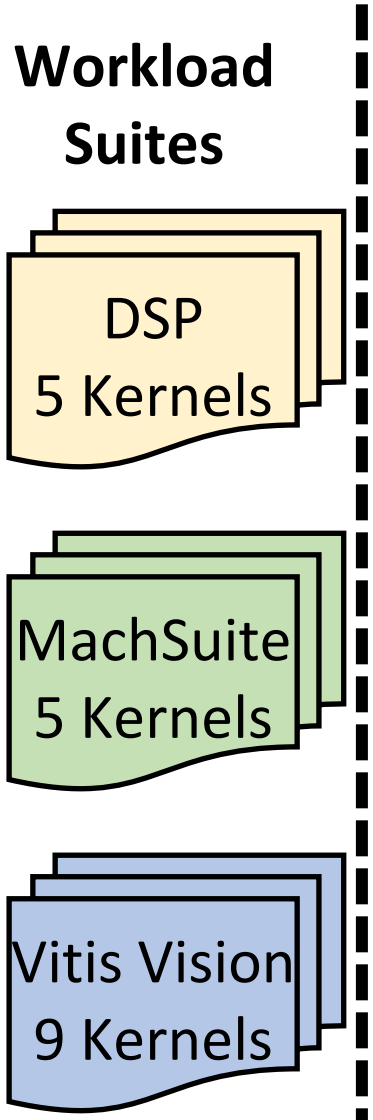
Object=Performance/Area

- Performance
  - Spatial architecture essentially enables hardware specialized sw-pipelining
  - The ratio of data availability determines the performance
  - Perf=#Inst * (Activity Ratio)
    - Insts offloaded
    - Stream
    - Memory B/w
    - …
- Area
  - ASIC: A regression model can predict the area number
  - FPGA: Predict the occupied % of each available resource type
    - LUT, DSP, FF, BRAM, SRL
    - Perf/max(% Occupied)

Roadmap

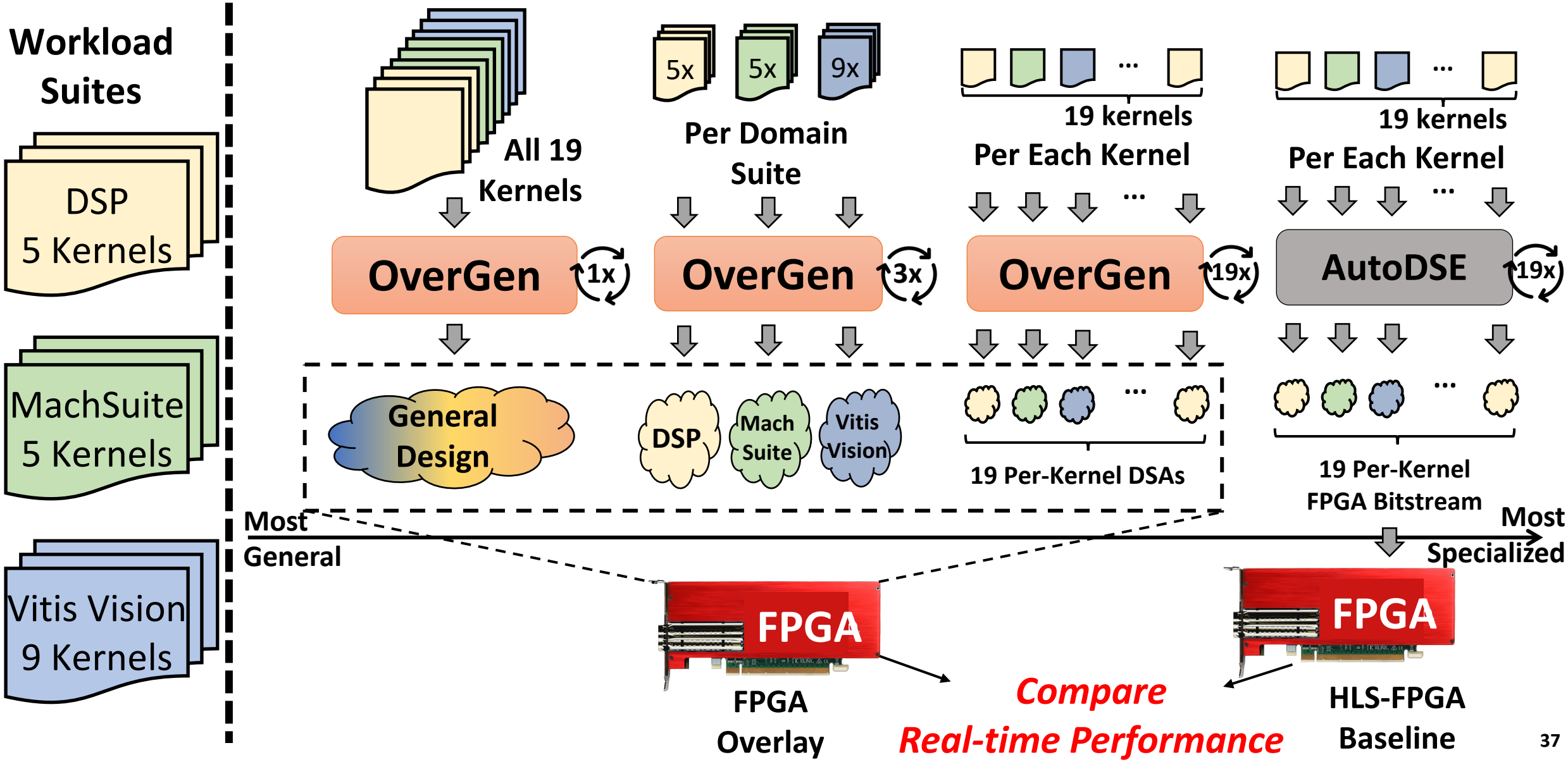
- Motivation
- Background
- Compiler
- Design Space Exploration
- **Evaluation**
- Future Work

Methodology

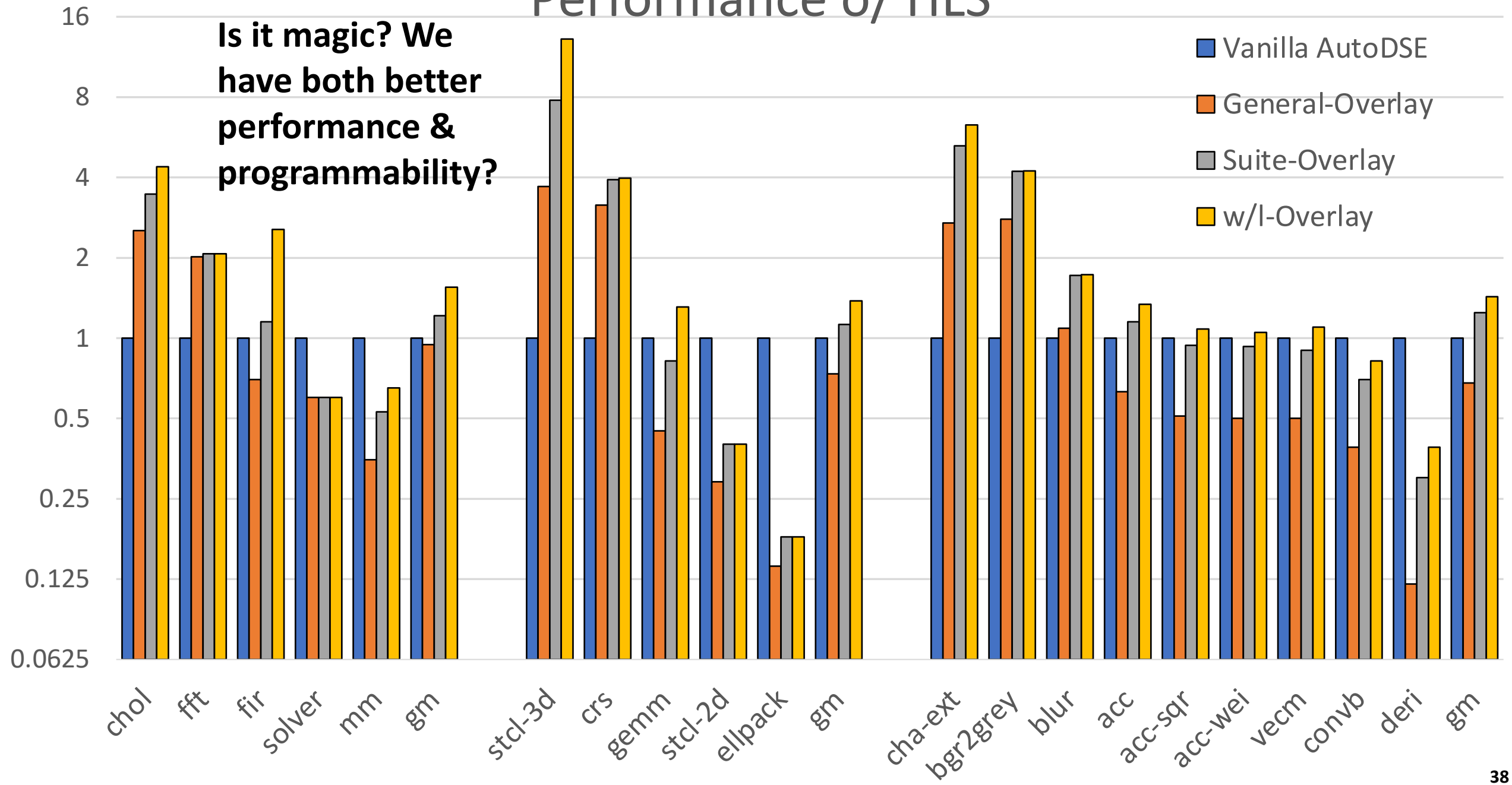


- Workloads
  - MachSuite: A benchmark suite for accelerator design
  - DSP: A set of inductive matrix algorithms
  - Vitis: Xilinx HLS workloads
- Baseline
  - AutoDSE: A SOTA Automated HLS Framework
- RTL Impl
  - Chisel for ADG RTL generation
  - Integrated to Chipyard & FireSim
- Performance
  - VCS Simulation

A Knob to Generality-Performance Tradeoff



Performance o/ HLS



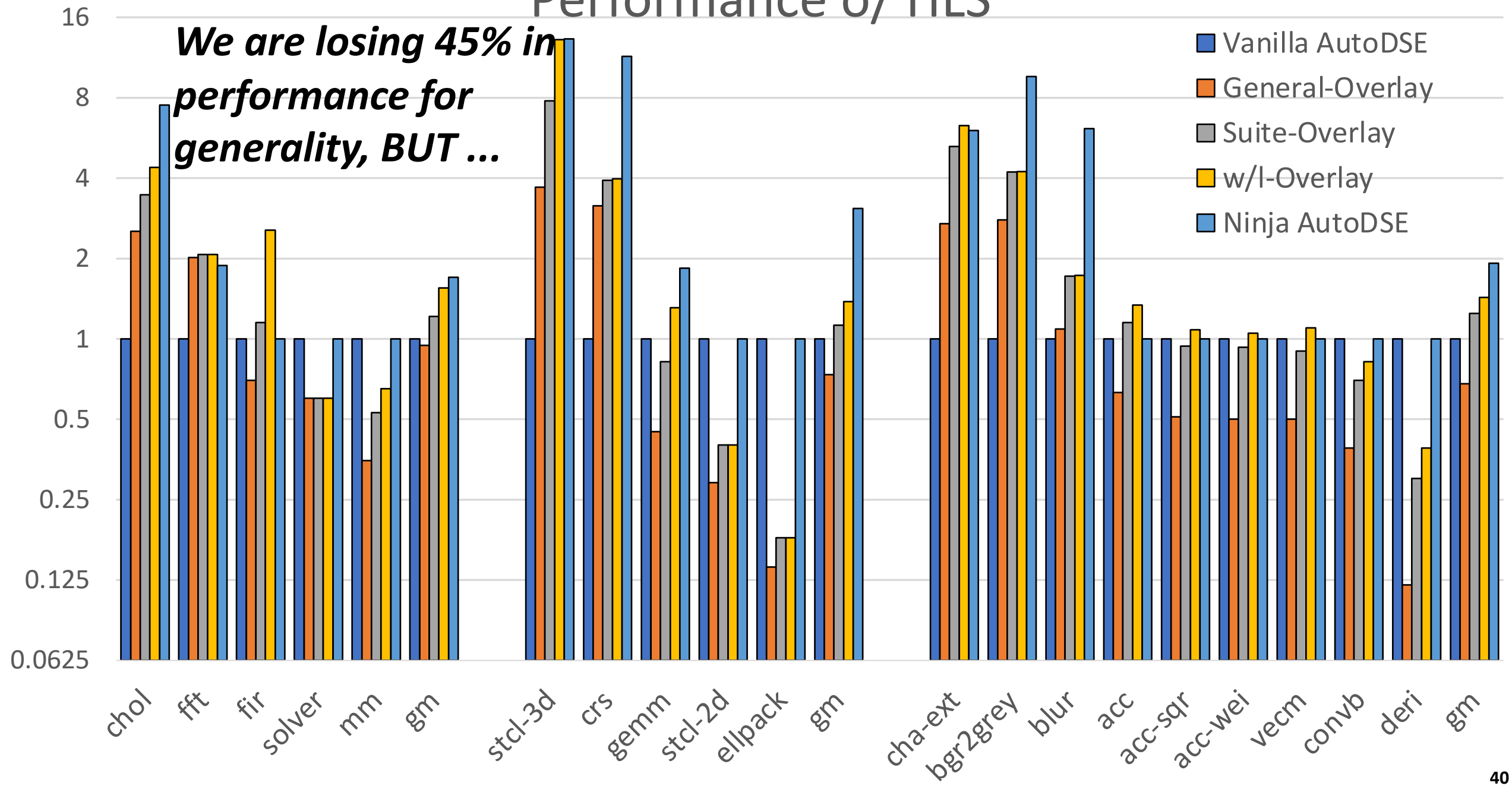
HLS-Oriented Tuning

- HLS generates state machine for inner-most loop body
  - Perfect loop body is preferred
  - Even “if” is prefered

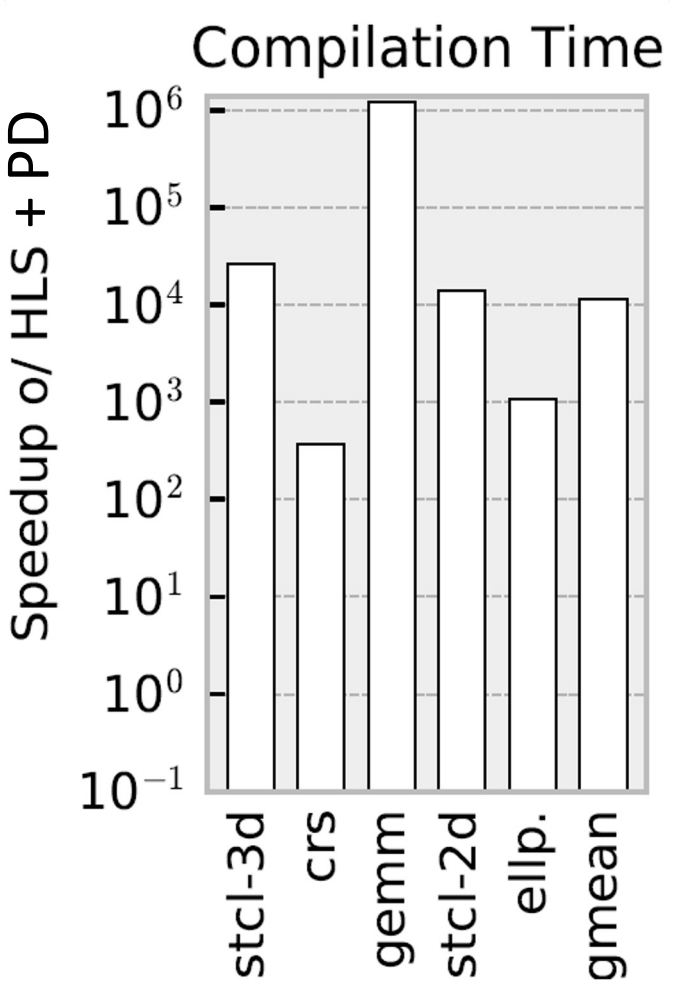
```
// Vanilla Code
for (int i=0; i<n; ++i) {
  // do something-1
  for (int j = 0; j<i; ++j)
    // do something-2
}
```

```
// HLS Prefered
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    if (j==0) {
      // do something-1
    }
    if (j<i) {
      // do something-2
    }
  }
}
```

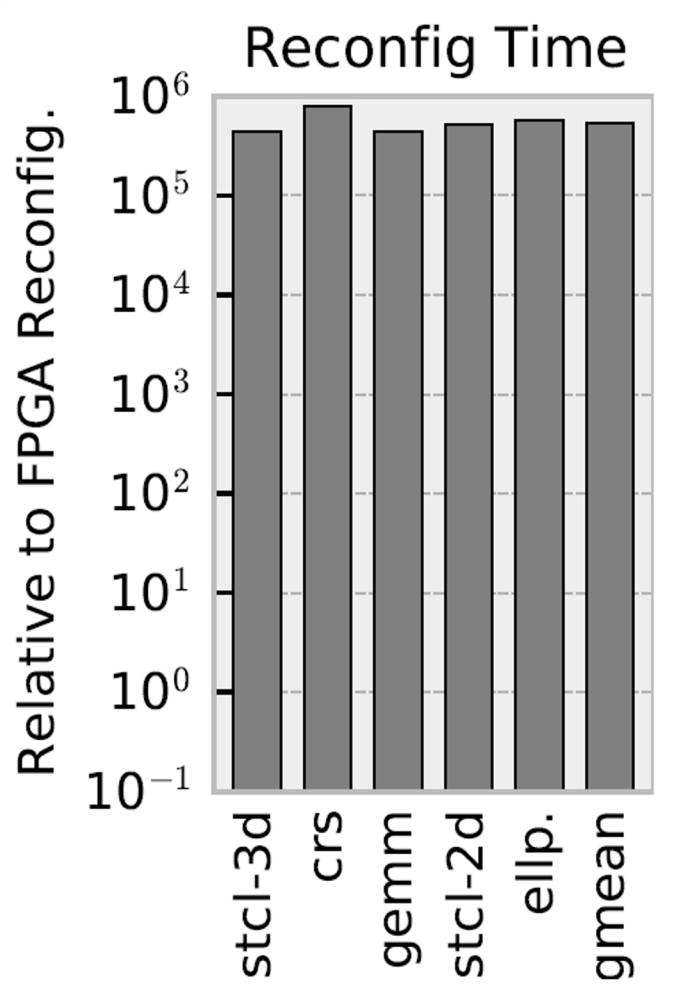
Performance o/ HLS



Compilation & Reconfiguration



10,000x faster in re-compile!



100,000x faster in reconf.!

Conclusion: Open-Source, Full-Stack E2E!

- Can be a challenger to existing FPGA programming paradigm
  - Comparable performance to HLS
  - Fast reconfigurable and time-multiplexing
  - More programmer-friendly
- Infrastructure available at
  - https://github.com/PolyArch/dsa-framework

Roadmap

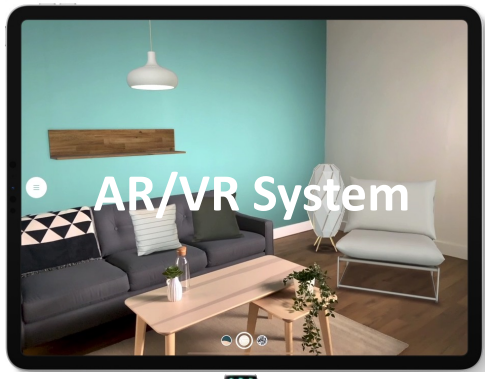
- Motivation
- Background
- Compiler
- Design Space Exploration
- Evaluation
- **Future Work**
  - **Revisit What I Left Last Time**
  - **Extending the Applicability**
  - **OS Management**

Ongoing Work (On Jan 27th, 2022)

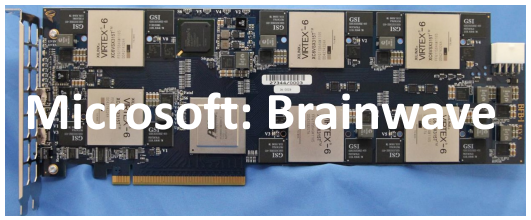
- ~~Design Space Extension~~
  - ~~Multicore Scale-up~~
  - Asymmetric Cores
  - ~~Synthesizing Memory Hierarchy~~
- ~~FPGA Prototyping~~

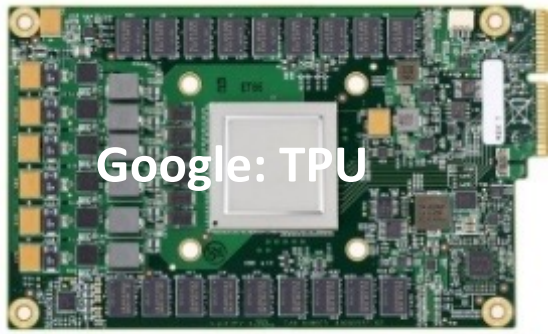
Extending Applicability to More Contexts

- Objective: Perf/Area
- Accelerators for different contexts require different objectives



Real-time System:
Latency (& Thermal)





Training in Data Center:
Energy Eff. & Throughput

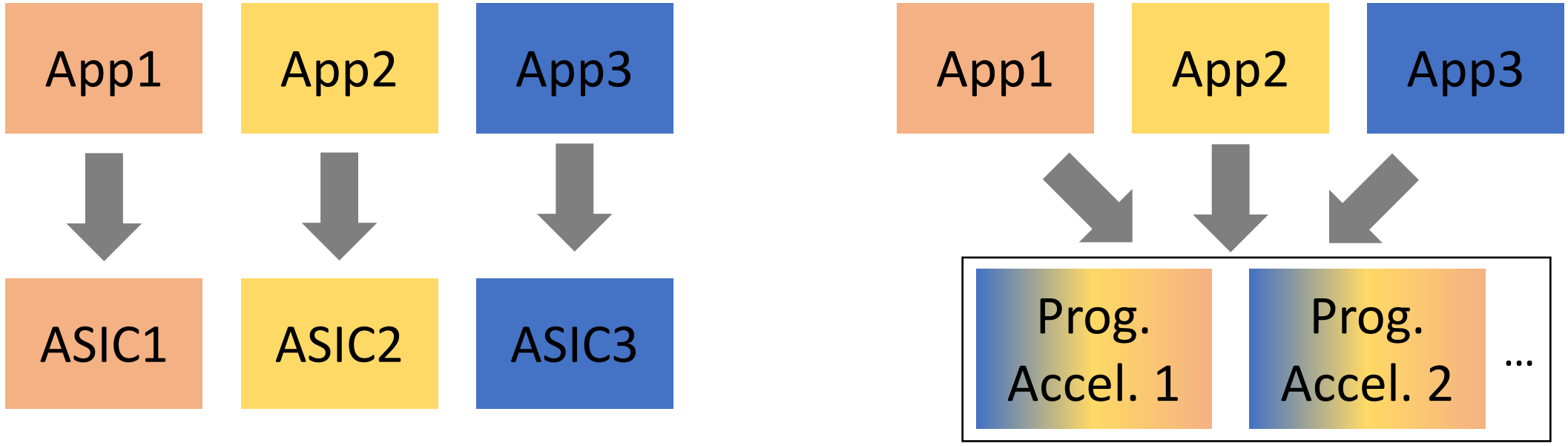
$$S = (1 - f) + \frac{f}{n}$$

n: Number of Cores
f: Fully Parallel

W/I can hardly scale up:
Single-core perf. & Frequency

- Research Questions：
  - How can we encode these requirements in objective functions?
  - How can we co-design specialized hardware mechanism and DSE strategies?

Sharing is a bless or curse?

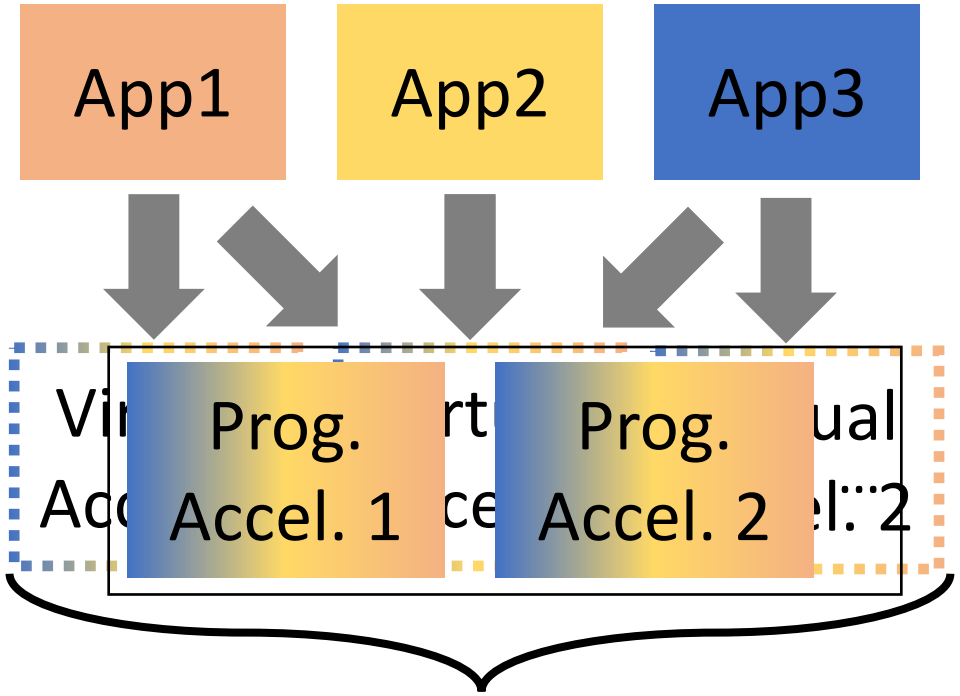


- How are applications assigned to hardware resources?
- How does the hardware deal with the concurrency?

Any problem in computer science can be solved by another layer of indirection (abstraction).

——Butler Lampson. Principles for Computer System Design. Turing Award Lecture. February 17, 1993

- s
- ment
- PEs
  - Dynamic: Tag-based isolation



Q&A: Thank you for listening!

- All the questions and discussions are welcomed!