Cadence Tensilica XNNC

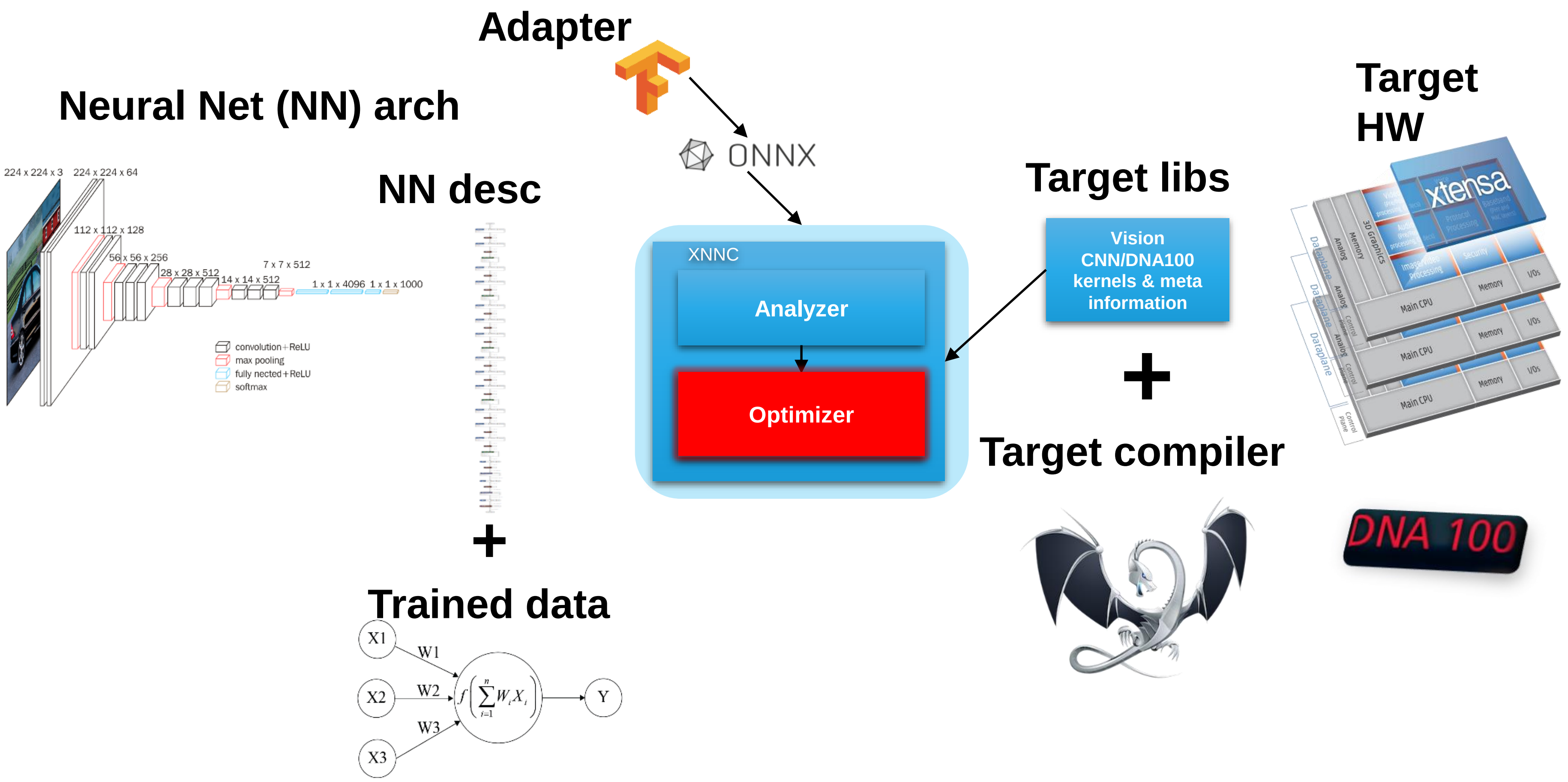
Xtensa Neural Network Compiler: Optimizer

Volodymyr Arbatov, Pedro Vaz Artigas, Xianmin Chen, et. al.
Presenter: Maxim Lukyanov

IEEE ML Compiler Workshop
San Jose, CA
March 4, 2019

XNNC: Overview

Xtensa Neural Network Compiler



Adapter

Neural Net (NN) arch

NN desc

+

Trained data

XNNC

Analyzer

Optimizer

Target libs

Vision CNN/DNA100 kernels & meta information

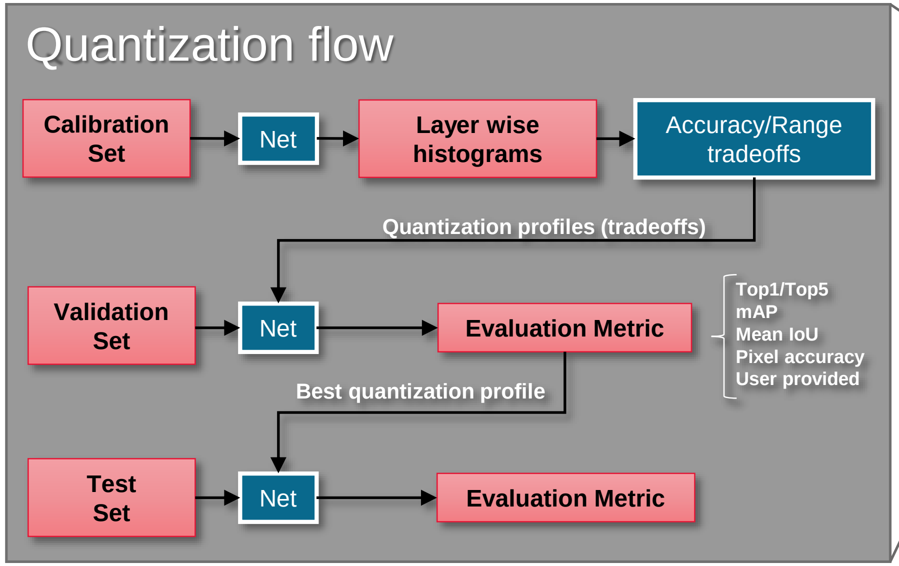
+

Target compiler

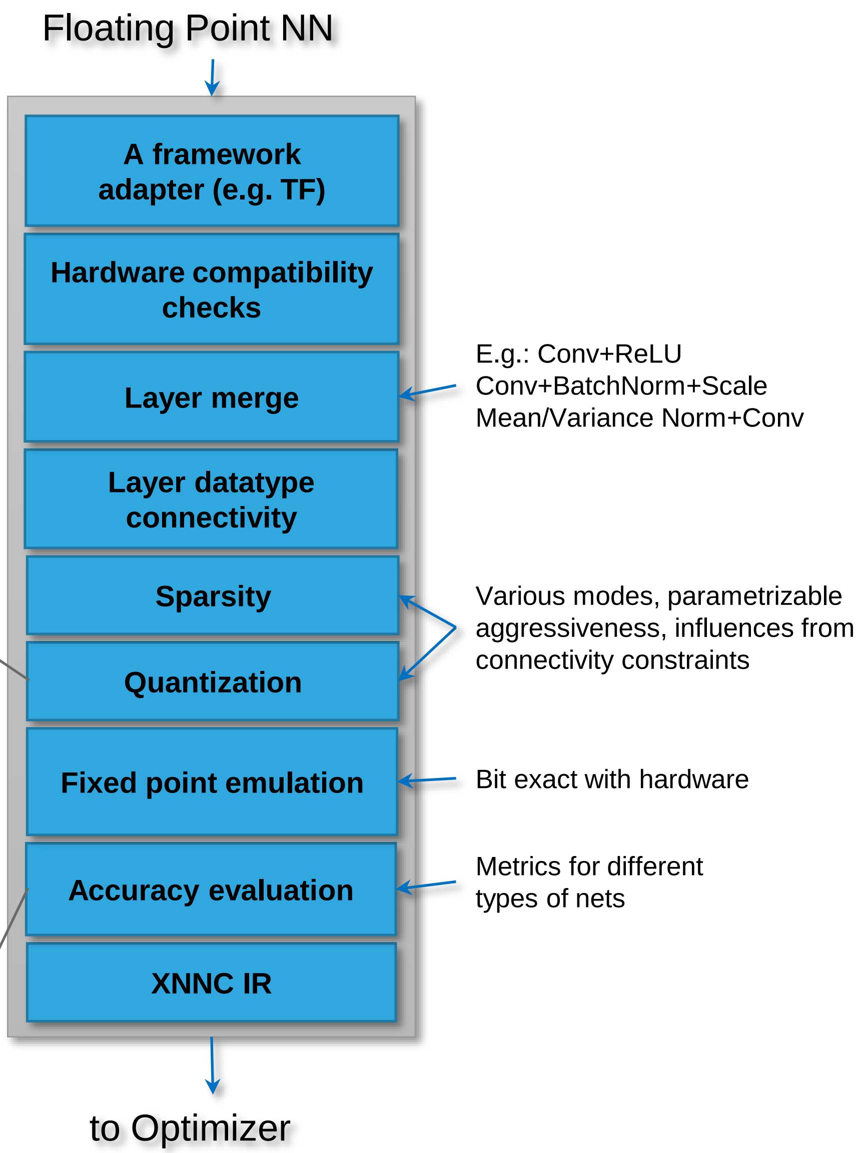
Target HW

XNNC: Analyzer
The front-end

- What's in it?
  - External framework support
  - Tensor manipulations
  - Network level optimizations & device-level pre-partitioning



Quantization flow



E.g.: Conv+ReLU
Conv+BatchNorm+Scale
Mean/Variance Norm+Conv

Various modes, parametrizable aggressiveness, influences from connectivity constraints

Bit exact with hardware

Metrics for different types of nets

XNNC: Optimizer

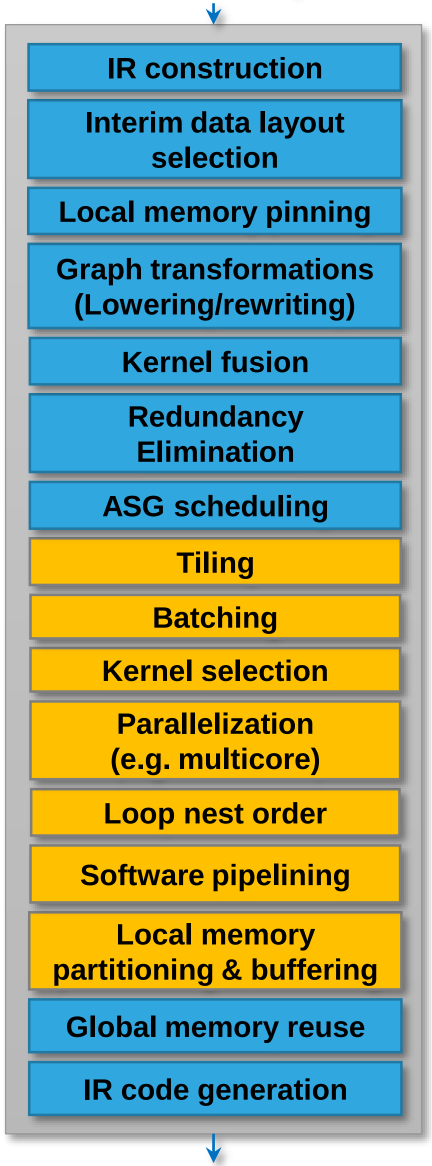
The middle-end

- Goal: Map a CNN from high-level pre-optimized IR into a lower-level representation with heavy-lifting optimizations

- Targets a family of Xtensa architectures:
  - DSPs: VP6, VC5, VQ6, VQ7, etc.
  - Accelerators: DNA100 and beyond…

- Recursive optimization process
  - Thousands of alternatives evaluated
  - Guided by performance estimations

IR + modified tensors
from the Analyzer



IR construction

Interim data layout selection

Local memory pinning

Graph transformations (Lowering/rewriting)

Kernel fusion

Redundancy Elimination

ASG scheduling

Tiling

Batching

Kernel selection

Parallelization (e.g. multicore)

Loop nest order

Software pipelining

Local memory partitioning & buffering

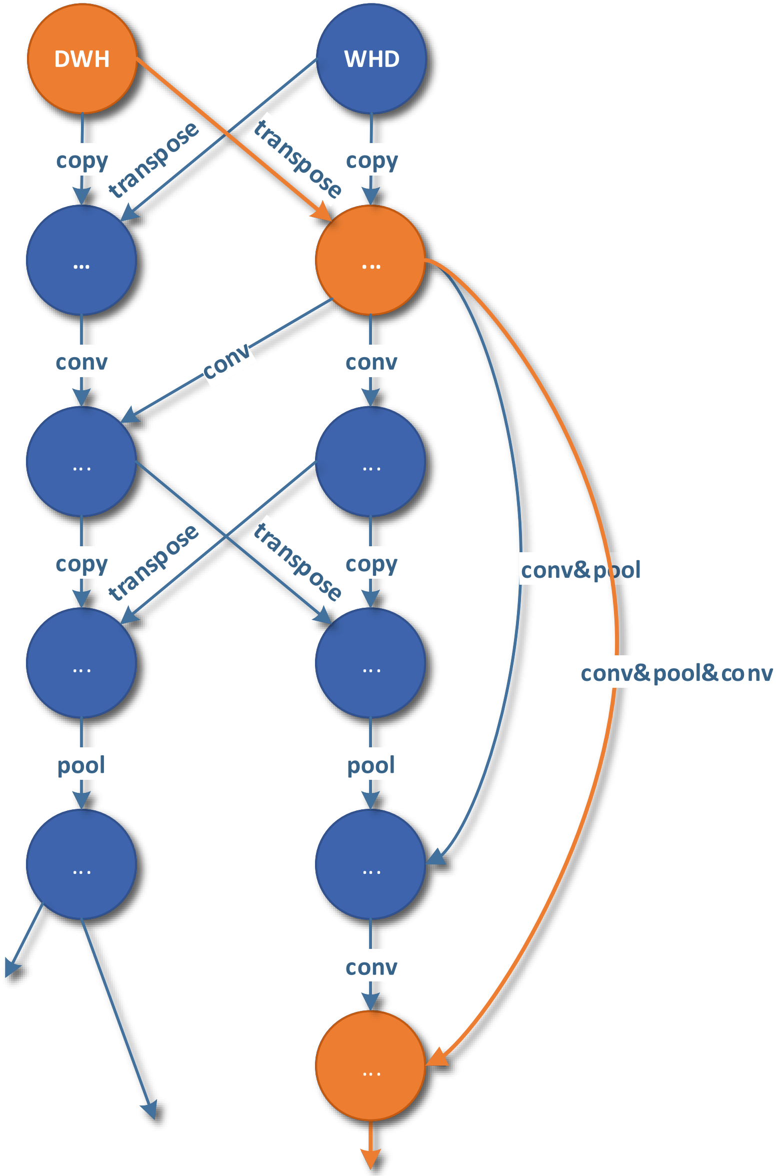
Global memory reuse

IR code generation

LIR or C or HAL to a back-end

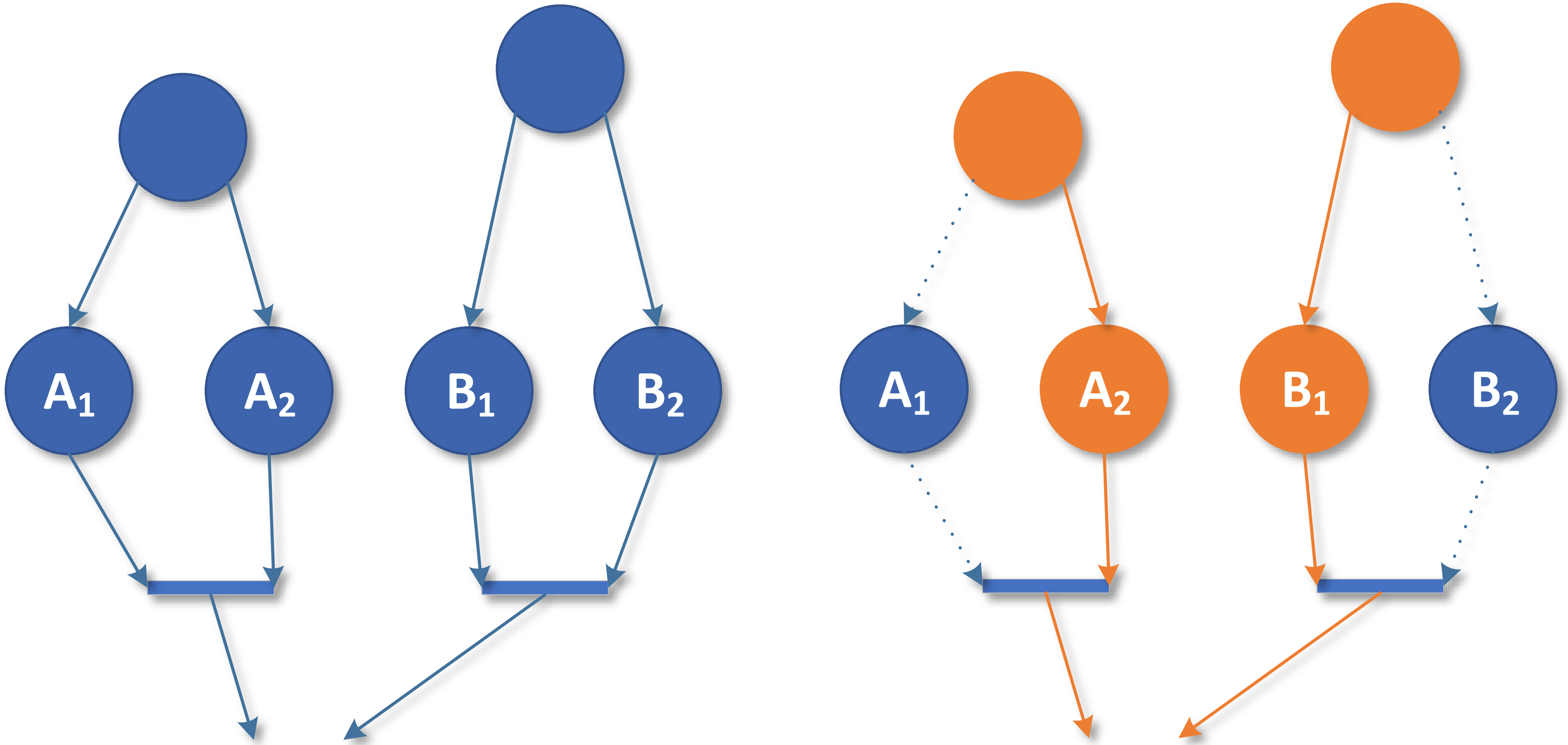
ASG: alternatives search graph
& interim layout selection

- Node – a data object global memory
- Edge – an operation (conv, pool, etc.)
- On construction:
  - Search space is heuristically capped
    - Contains *meaningful transitions* between data object formats
  - Augmented with weights (perf model)
- As we go: lower/expand, redundancy elimination, local edge re-writing
- Goal: Find a fastest performing subgraph among alternative solutions
  - Dijkstra w/ modified relaxation criteria in presence of "meet" nodes



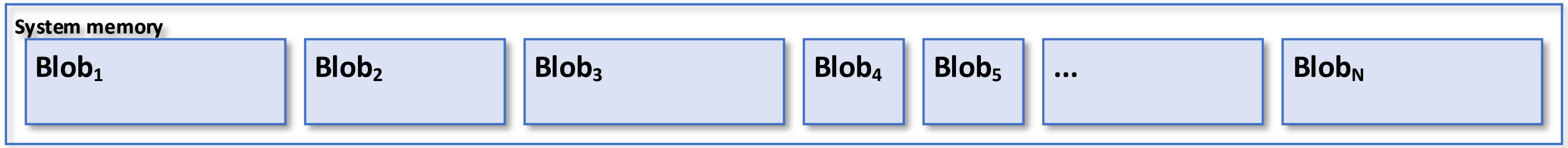
ASG: More on alternatives…

- Ex: *eltWise add* – 2 inputs, each supplied in a number of alternative layouts from predecessor kernels

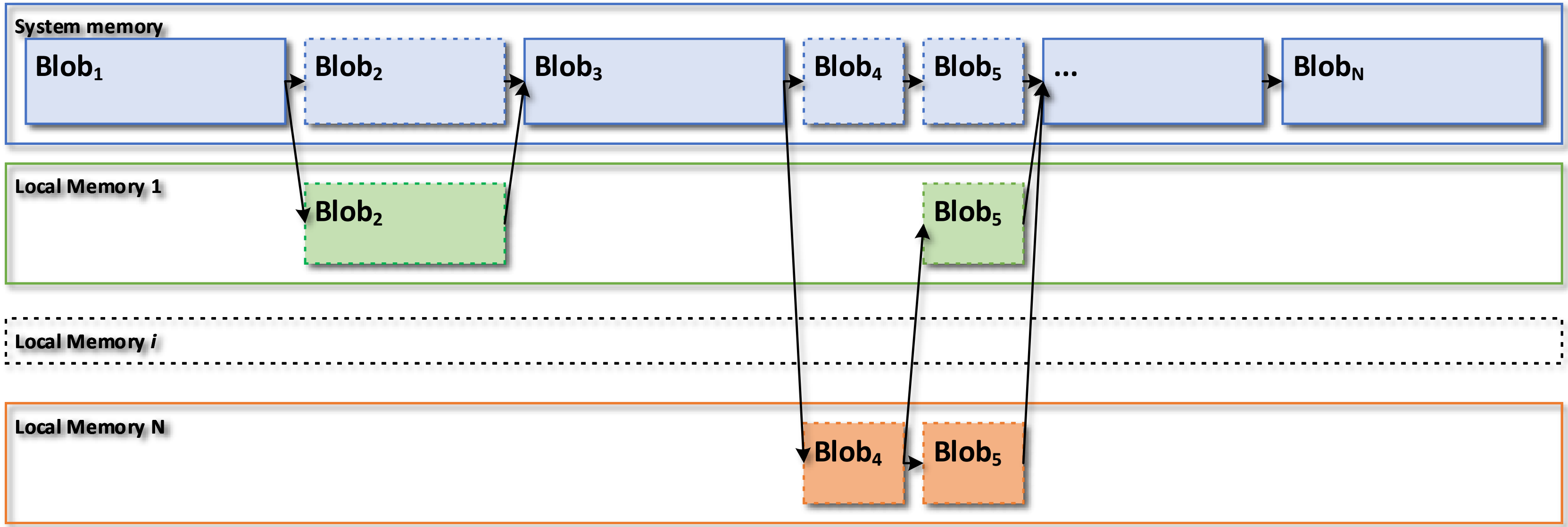


ASG: pinning objects to local memory

- Initial

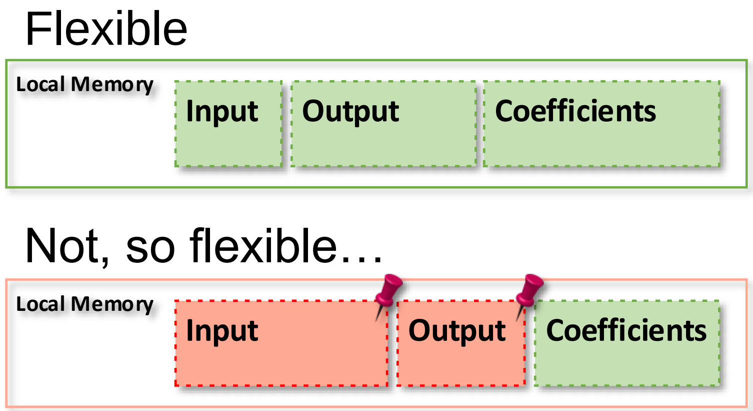


- After pinning to a hierarchy



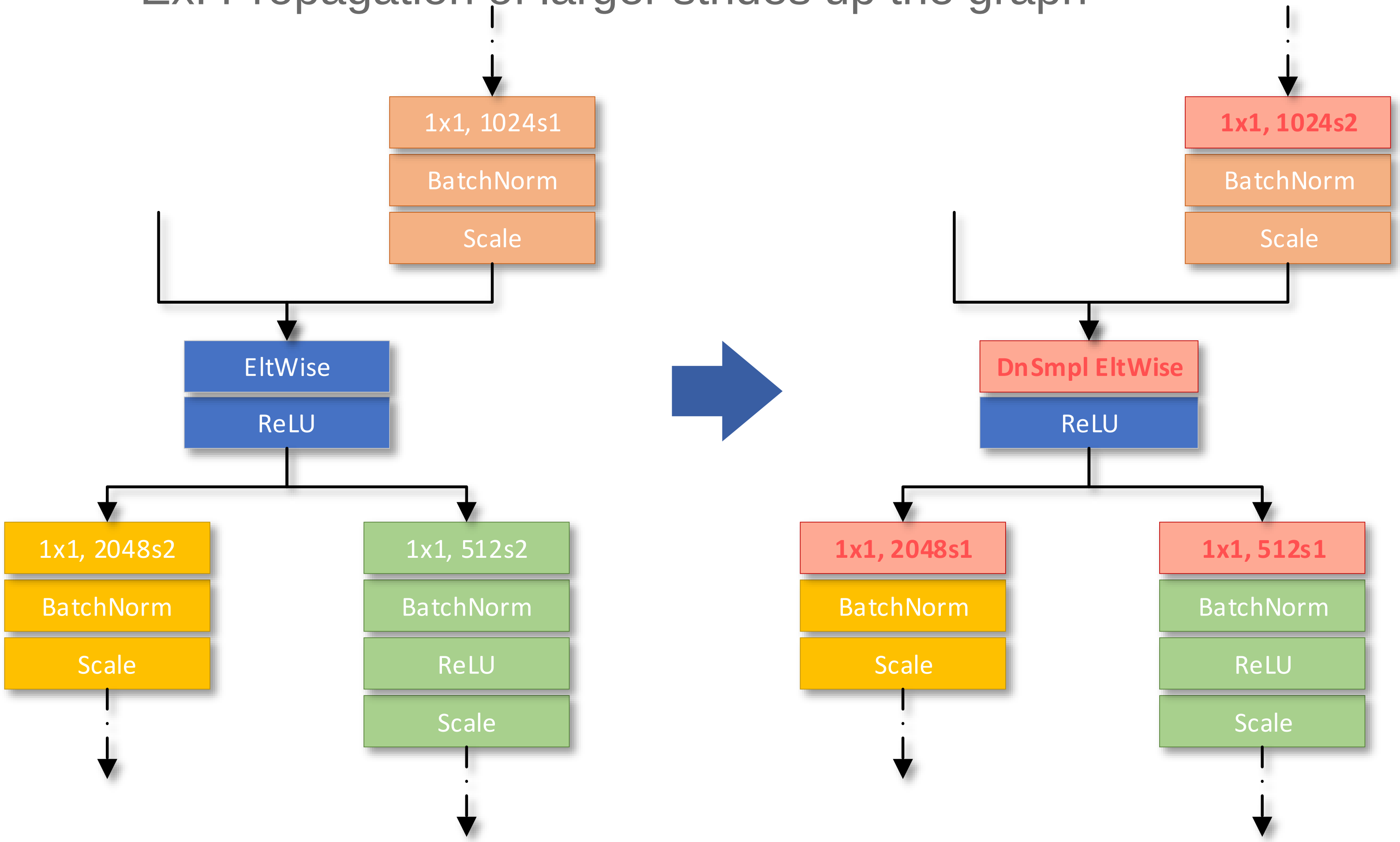
ASG: pinning objects to local memory

- Extend ASG with nodes representing local memory objects
  - Only consider objects that fully fit
- Note: decision is made via ASG – not a heuristic!
- Take away:
  - Not always efficient to keep intermediate outputs in local memory
  - May lead to:
    - Smaller tiles, more loop overhead and DMA ops
    - Poorer efficiency of computation kernels
      - Kernels – general term can be ISA, fixed function or library functions
    - Overfetch (ex: reloading coefficients)



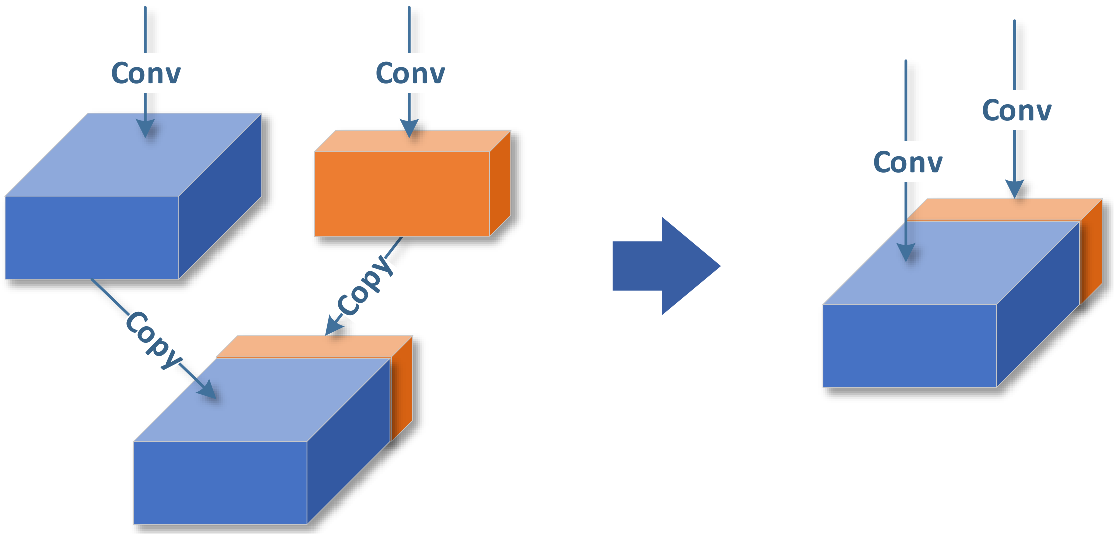
ASG: redundancy: strength reduction

- Redundant compute:
  - Ex: Propagation of larger strides up the graph



ASG: Redundancy: copies and layouts

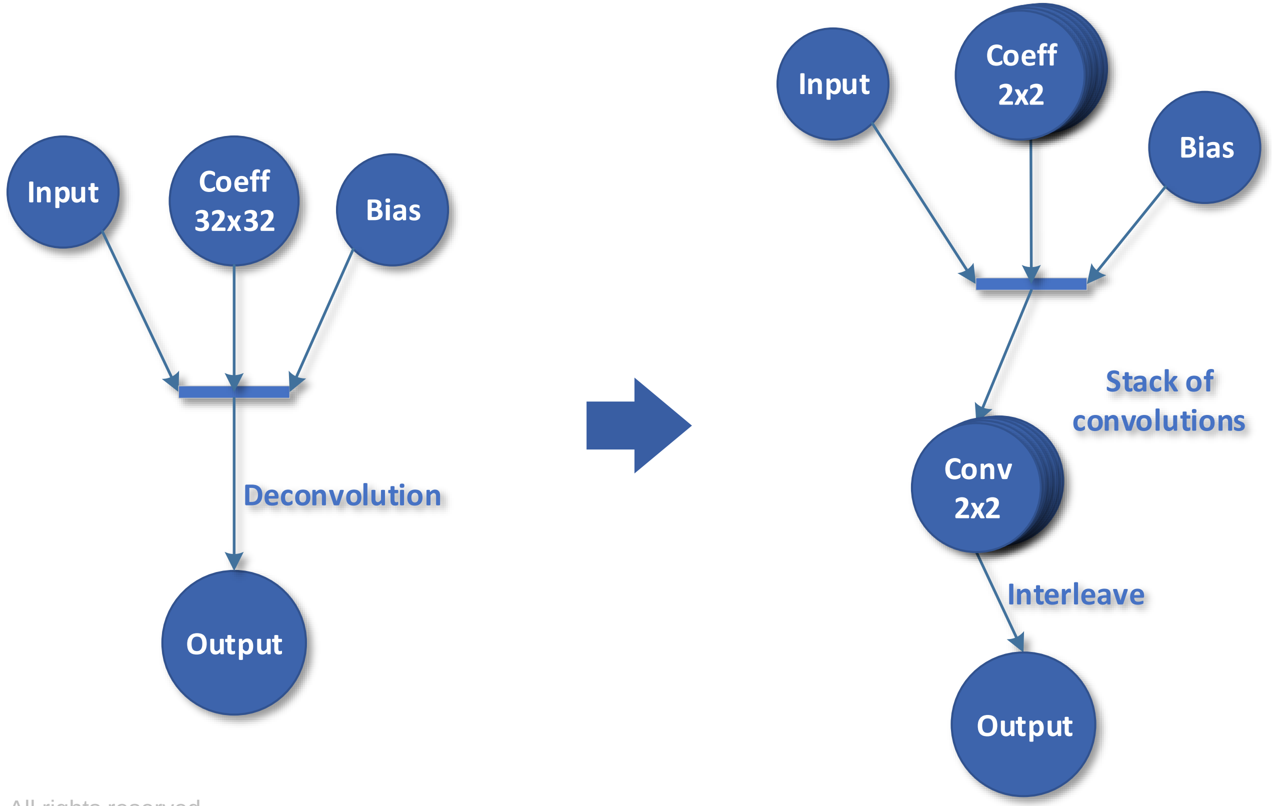
- Most spurious copies introduced for connectivity reasons go away
- Ability to utilize non-linear memory operations for copying
  - Ex: depth-wise concat in place
- Fuse aggressively to minimize memory objects in a given path
- Transform layout of data objects to allow more efficient memory transfers and computations
  - Example: Pad a tensor to conform to memory access alignment rules



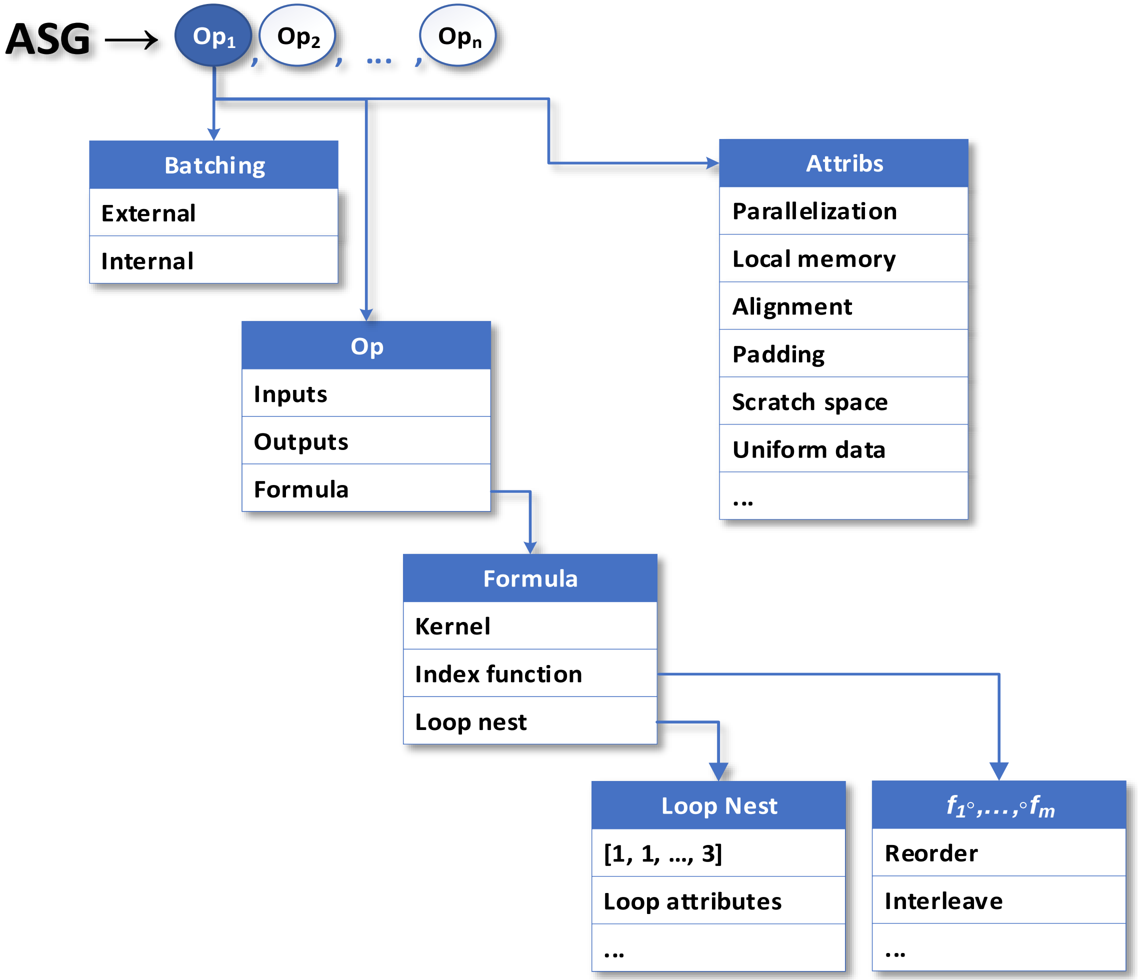
cadence®

ASG: Edge rewriting and operations expansion

- Graph transforms/lowering:
  - A number of pattern matching rules
  - Ex: deconvolution is expanded into a set of convolutions + an interleave
    - Interleave can be done in local storage or DMA (i.e. may later disappear)



XNNC IR: 10K foot view



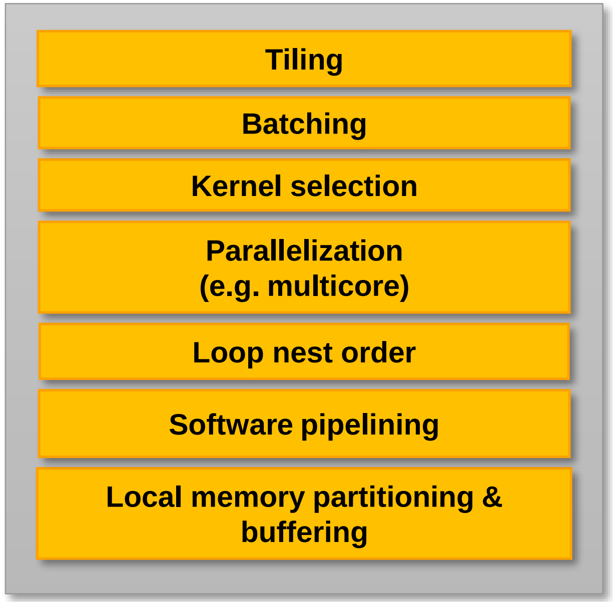
- Multistage IR
  - Operations are subject to stage-based rule rewriting
  - A rule is an transformation or annotation of the IR
  - Explicit semantics for high level operations
    - Rules applied “gnostically”
  - Loop abstractions – first class citizens
  - Converges to straightforward unambiguous code generation

XNNC IR: At the start of compilation

```
BATCH( input   = [ax:DATATYPE0:whd:[227:227:3] pitch:[1:227:51529]]
       output  = [ay:DATATYPE2:dwh:[1000:1:1] pitch:[1:1000:1000]]
       count   = 4
       ## Formula ##
       dagraph( name   = 'formula',
                input  = [ax:DATATYPE0:whd:[227:227:3] pitch:[1:227:51529]]
                output = [ay:DATATYPE2:dwh:[1000:1:1] pitch:[1:1000:1000]]
                op( name    = 'conv1',
                       input   = [ax:DATATYPE0:whd:[227:227:3] pitch:[1:227:51529],
                                  conv1_coeff:DATATYPE0:whdn:[11:11:3:96] pitch:[1:11:121:363]
                                      DATA: dRandom(1, DATATYPE0:whdn:[11:11:3:96] pitch:[1:11:121:363], -128, 127),
                                  conv1_bias:DATATYPE4:bias:[96] pitch:[1]
                                      DATA: dRandom(2, DATATYPE4:bias:[96] pitch:[1], -128, 127)]
                       output  = [t1:DATATYPE3:dwh:[96:55:55] pitch:[1:96:5280]]
                       ## Formula ##
                       cnnConv3D((0, 0, 0, 0), [11, 11, 3], (4, 4), 1, 1, True, {'PARAM1’ … 'PARAM10'}, None, False, None)
                ),
                op( name    = 'norm1',
                       input   = [t1:DATATYPE3:dwh:[96:55:55] pitch:[1:96:5280],
                                  datastructure0:DATATYPE1:[1024] pitch:[1]
                                      DATA: dRandom(3, DATATYPE1:[1024] pitch:[1], 0, 32767)]
                       output  = [t2:DATATYPE0:dwh:[96:55:55] pitch:[1:96:5280]]
                       ## Formula ##
                       cnnLRN((5, 5), cnnLRN, {'PARAM1’, … ,'PARAM5'})
                ),
                op( name    = 'pool1',
                       ......
                ),
                ......
                op( name    = 'prob',
                       input   = [t13:DATATYPE1:dwh:[1000:1:1] pitch:[1:1000:1000],
                                  datastructure2:DATATYPE2:[9875] pitch:[1]
                                      DATA: dRandom(19, DATATYPE2:[9875] pitch:[1], 0, 65535)]
                       output  = [ay:DATATYPE2:dwh:[1000:1:1] pitch:[1:1000:1000]]
                       ## Formula ##
                       cnnSoftmax(D, {'PARAM1’, 'PARAM2'})
                )
```

XNNC IR: Local transformations and rules

- Applicable to a single operation in the search graph and applied recursively:
  - Each local transformation has a set of rules associated with it
  - Top level transforms are invoked in-order each examining degrees of freedom per rule
    - A degree of freedom is typically associated with a parameter, e.g. *tile size*
  - If Transform *i* explored *N* degrees of freedom applied to a single parameter, it'll yield *N* variants to Transform *i+1*
  - Given *M* parameters per rule, with each having a freedom degree it's a vast space, obviously growing exponentially
  - When exploring the search space various heuristics kick in to cap traversal of the space



Tiling

Batching

Kernel selection

Parallelization
(e.g. multicore)

Loop nest order

Software pipelining

Local memory partitioning &
buffering

- Although order is initially fixed, each transformation may yield some or no opportunities for subsequent ones

XNNC IR: Introducing loop nests

- Tiling – introduces an initial loop nest structure with default index maps.
- Index mapping function:
  - Basically, an affine map for data and iteration spaces

```
[ ## Gathers ##
  ....
  H(  domain: DATATYPE0:align=X:ndwh:[((ind(2, 7) < 7-1)?(14):(12)):3:11:11]  # Ex: Local memory
      range:  DATATYPE0:align=X:ndwh:[14:3:11:77]                             # Ex: Global memory
      [ # offs dom.n dom.d dom.w dom.h ..loops..
        [0, 1, 0, 0, 0, 0, 0, 0,  0] # rng.n, pitch: 1
        [0, 0, 1, 0, 0, 0, 0, 0,  0] # rng.d, pitch: 64
        [0, 0, 0, 1, 0, 0, 0, 0,  0] # rng.w, pitch: 192
        [0, 0, 0, 0, 1, 0, 0, 11, 0] # rng.h, pitch: 2112
      ]
  )]
[ ## Scatters ##
  H(  domain: DATATYPE3:dwh:[((ind(2, 7) < 7-1)?(14):(12)):55:((ind(1, 2) < 2-1)?(28):(27))]
      range:  DATATYPE3:dwh:[96:55:55]
      [ # offs dom.d dom.w dom.h ..loops..
        [0, 1, 0, 0, 0, 0, 14, 96] # rng.d, pitch: 1
        [0, 0, 1, 0, 55, 0, 0,  0] # rng.w, pitch: 96
        [0, 0, 0, 1, 0, 28, 0,  0] # rng.h, pitch: 5280
      ]
  )
]],
 ## Loops ##
[1, 2, 7, 1]
attribute loop_order: ALoopOrder(2, 1, 0, 3)
```

XNNC IR: Example of an index mapping function*

```
[ ## Gathers ##
    ....
    H(  domain: DATATYPE0:align=X:ndwh:[14:3:11:11]
        range:  DATATYPE0:align=X:ndwh:[14:3:11:77]
        [ # offs dom.n dom.d dom.w dom.h ..loops..

            [0, 1, 0, 0, 0, 0, 0, 0,  0] # rng.n, pitch: 1
            [0, 0, 1, 0, 0, 0, 0, 0,  0] # rng.d, pitch: 64
            [0, 0, 0, 1, 0, 0, 0, 0,  0] # rng.w, pitch: 192
            [0, 0, 0, 0, 1, 0, 0, 11, 0] # rng.h, pitch:
2112        P           S
        ]
    )
],
[ ## Scatters ##
    H(  domain: DATATYPE3:dwh:[14:55:28]
        range:  DATATYPE3:dwh:[96:55:55]
        [ # offs dom.d dom.w dom.h ..loops..

            [0, 1, 0, 0, 0, 0, 14, 96] # rng.d, pitch: 1
            [0, 0, 1, 0, 55, 0, 0,  0] # rng.w, pitch: 96
            [0, 0, 0, 1, 0, 28, 0,  0] # rng.h, pitch: 5280
            P           S

        ]
    )
]],
 ## Loops ##
[1, 2, 7, 1]
attribute loop_order: ALoopOrder(2, 1, 0, 3)
```

Strided index mapping:

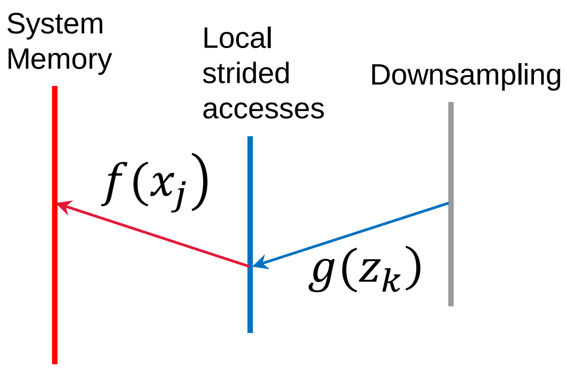
$$f\langle x_1, \dots, x_n\rangle \rightarrow y_1, \dots, y_m$$

$$y_i = P_i + \sum_{j=1}^{n} x_j S_{ij}$$

Strided index mapping composition (at compile-time):

$$g\langle z_1, \dots, z_l\rangle \rightarrow x_1, \dots, x_n$$

$$f \circ g \rightarrow f\langle g\langle z_1, \dots, z_l\rangle\rangle \rightarrow y_1, \dots, y_m$$



* - For details, see: *F. Franchetti et al., "SPIRAL: Extreme Performance Portability," in Proceedings of the IEEE, vol. 106, no. 11, pp. 1935-1968, Nov. 2018.*

XNNC IR: Kernel selection

- Kernels – generic term for us to describe an atomic entity or a primitive
  - Can be an ISA intrinsic, a fixed function unit or a library function
- Kernel descriptions carry attributes
  - represented either computationally or via a composition of index mapping functions
- We narrow a set of kernels applicable to an operation with a number parameters
  - data layout, data types, I/O sizes, strides/dilations, etc.
- Ultimate choice is based on a set of evaluations based on the XNNC performance model (using the same recursive scheme)

```
dagraph( name   = 'gt',
         ....
         layer( name    = 'gt',
                input   = [input0:  ..., input1:  ..., input2:  ...,]
                output  = [output0: ... ]
                ## Formula ##
                xiConvolved3D_S_MxN_DATATYPE0DATATYPE0IXCa2_MOD_WHD_DWH(
                IOPorts([DATATYPE0:whd:[217<5,5>:((ind(1, 2) < 2-1)?(109):(105))<5,5>:3],
                         DATATYPE0:align=X:ndwh:[((ind(2, 7) < 7-1)?(14):(12)):3:11:11],
                         DATATYPE4:bias:[((ind(2, 7) < 7-1)?(14):(12))] pitch:[1]],
                        [DATATYPE3:dwh:[((ind(2, 7) < 7-1)?(14):(12)):55:((ind(1, 2) < 2-1)?(28):(27))], []),
                        [11, 11, 3], (4, 4), 1, True, None, {'PARAM1' ...  'PARAM10'})
         )
)
```

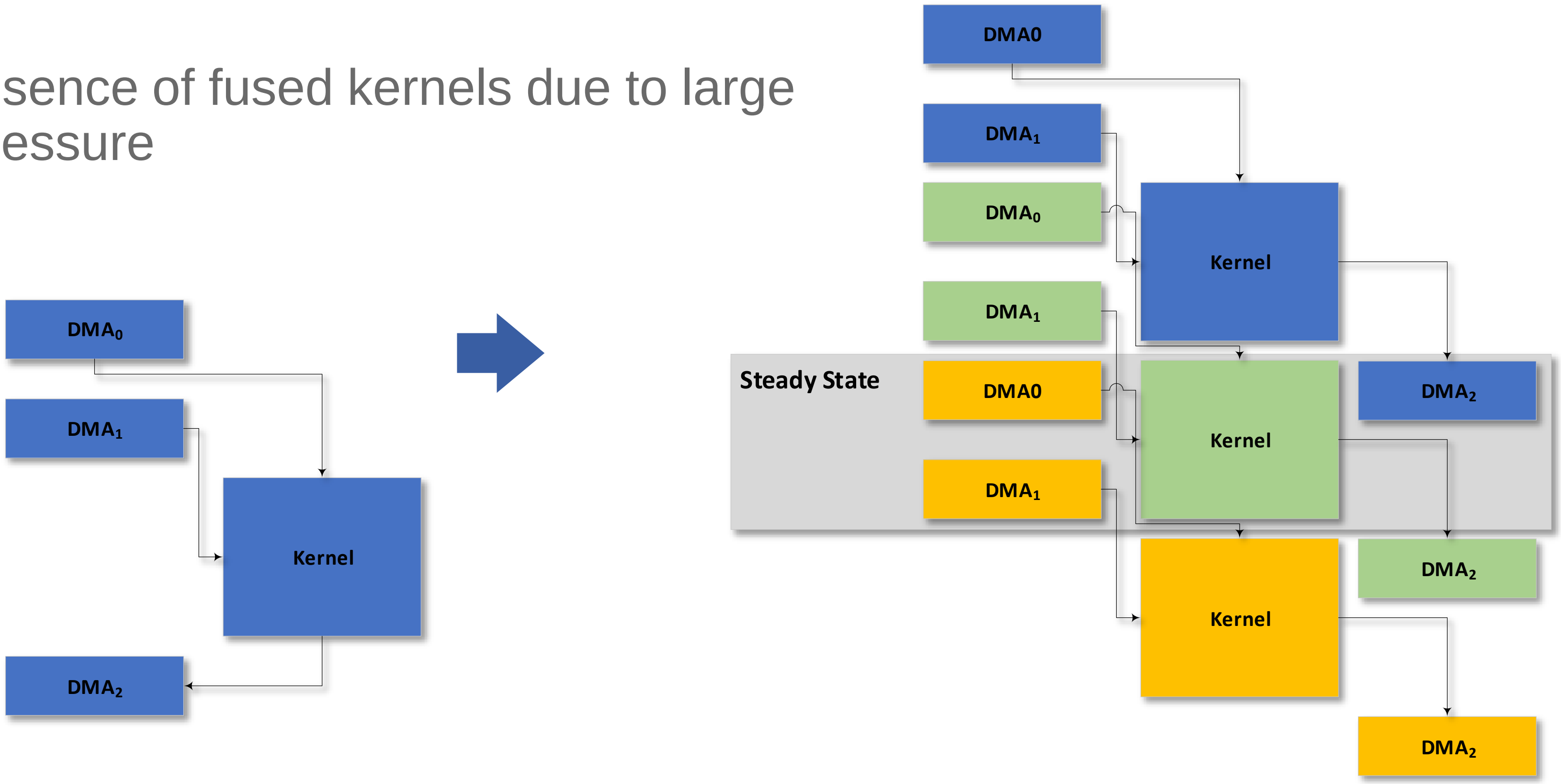
XNNC IR: Parallelization/batching & kernel fusion

- Parallelization and batching:
  - **Parallelization** – effectively picking a loop in a loop nest and performing a loop interchange to make it outermost
    - Can be nested, if hierarchy’s necessary, i.e. vectorization
  - **Batching** introduces a loop over batch of elements
    - Merge it with a loop nest for an operation
    - Easy, just an extra loop dimension!
    - Can also be mapped onto a kernel that handles batching

- Kernel fusion
  - Ex: Fusing 2 convolutions
    - Tile the last convolution first
    - The index function that gathers elements for the bottom convolution will determine the shape of the top convolution output
    - Continue propagate information up to the input of an operation as a whole
    - Intermediate data is kept in local memory
    - The rest is identical to forming any other loop nest

XNNC IR: Software pipeliner

- Main goals:
  - Execution of DMA transfers & heterogenous kernels pipelined
  - Local memory partitioning and buffering
  - Using multi-buffering schemes
- Takeaway:
  - Gets funky very fast in presence of fused kernels due to large IIs and increased buffer pressure



ASG: Final touches

- ASG scheduling:
  - Given a dataflow subgraph of ASG – schedule it
  - The criteria for a topological sort is reduction of storage pressure
    - I.e. minimize the number of buffers that are simultaneously live

- Storage coalescing:
  - Decide which storage is shared by which data objects
  - A typical allocation problem
  - Uses a technique similar to linear scan register allocation sans spilling

Performance model

- >500 DSP kernels used by the compiler to select from
  - More if we consider accelerators
- Each has parameterized model derived from its code structure
  - Covers all DMA transfers, loop nest structure, latency, access patterns for all kernels
- Very high accuracy

Network	Error
AlexNet	2.5%
Inception V3	1.6%
ResNet 50	3.7%
MobileNet V1	1.4%

- Why do we need an accurate performance model?
  - To generate code, compile, and perform a cycle accurate simulation run: usually takes about 10 seconds per measure per sample
  - Very slow if thousands and thousands of samples!
- Parameters of the performance model of a kernel are built automatically from large number of samples taken during search.

XNNC: Automatic verification

- Simplified reference is generated along with a net binary:
  - high level, bit exact reference functions
- Automated testing of optimized code against reference
  - all necessary harness generated by the compiler

- We heavily use an option to apply transforms randomly
  - Doesn’t use perf model, just randomly transforms the code
  - Ex: random graph transformations, rules, attributes, tile sizes etc.
  - Produces valid implementation, verified against a reference
    - I.e. it “classifies” or “object detects” just may take a while
  - Greatly improves code coverage, may create unanticipated graph topologies.

XNNC Optimizer: what was our goal here?

- To mention the obvious:
  - Powerful framework with state-of-the-art techniques… sure!

- …but, most importantly:
  - High level of abstraction
    - Do not lower too soon, hard or impossible to backtrack
    - Semantic ties with multiple targets – know what you’re compiling for!
  - Unambiguous and straight-forward code generation
  - Ultra precise performance modelling
    - Critical for IP, don’t need HW to tune

Thank you!
Q&A