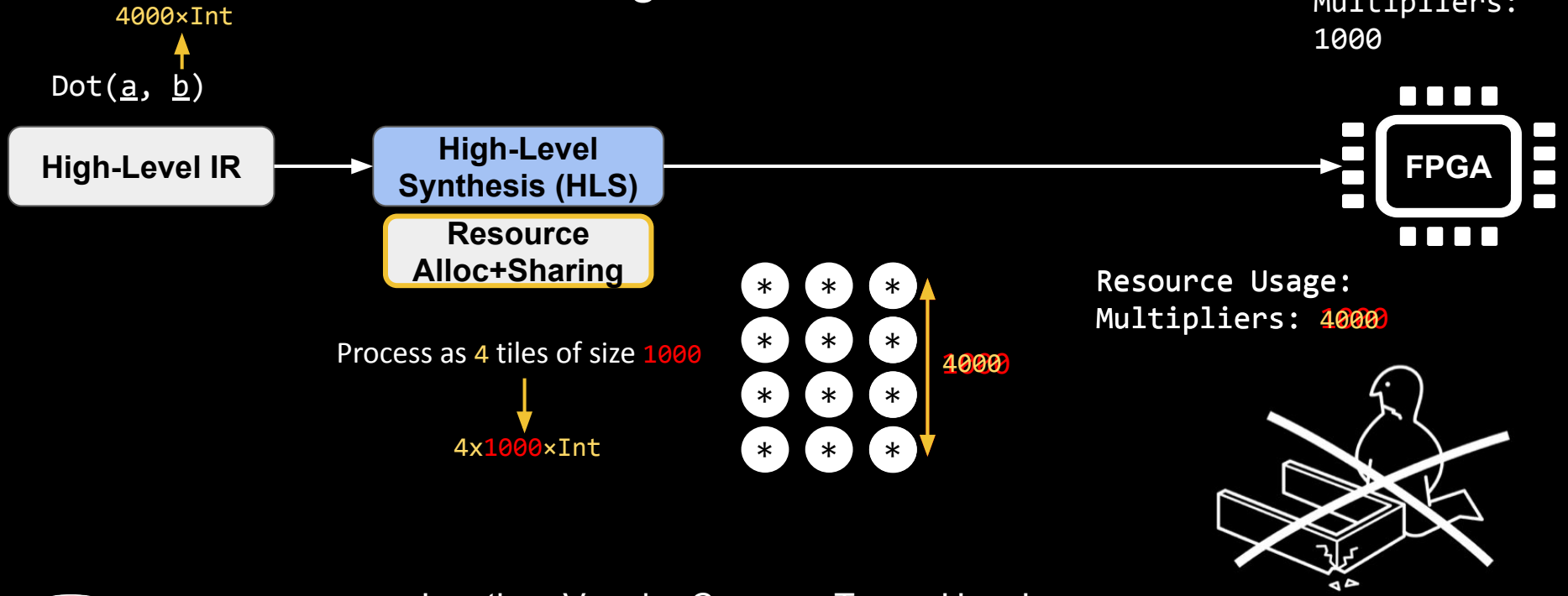
SkeleShare – Algorithmic Skeletons and Equality Saturation for Hardware Resource Sharing



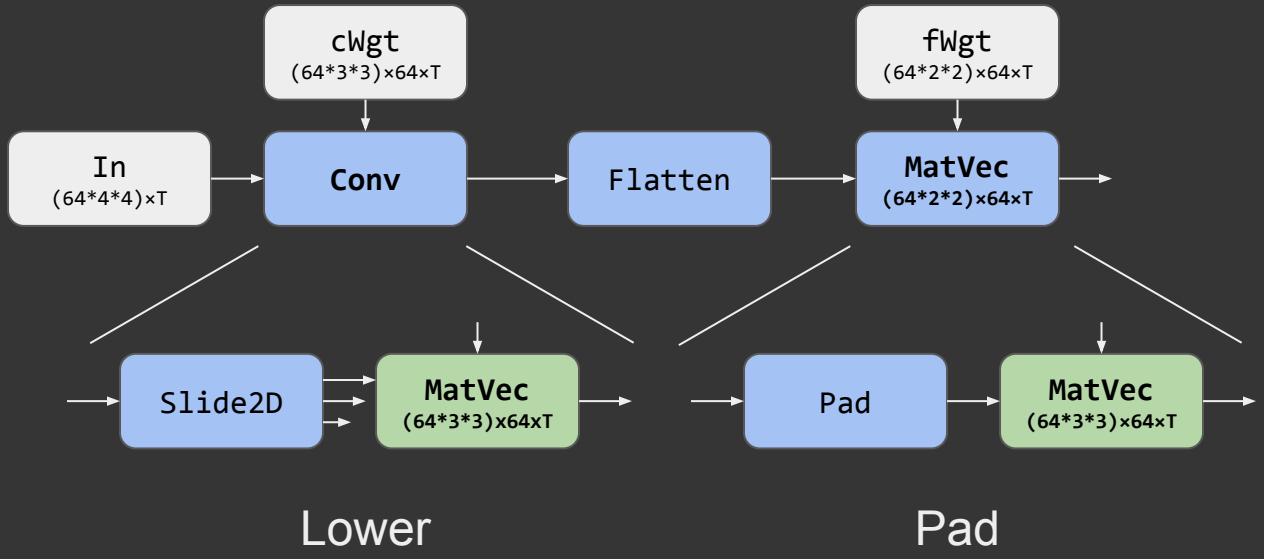
McGill

Jonathan Van der Cruysse, Tzung-Han Juang, Shakiba Bolbolian Khah, and Christophe Dubach

McGill University – CGO '26 – Monday February 2, 2026

Resource Sharing

`MatVec(fWgt, Flatten(Conv(cWgt, In)))`

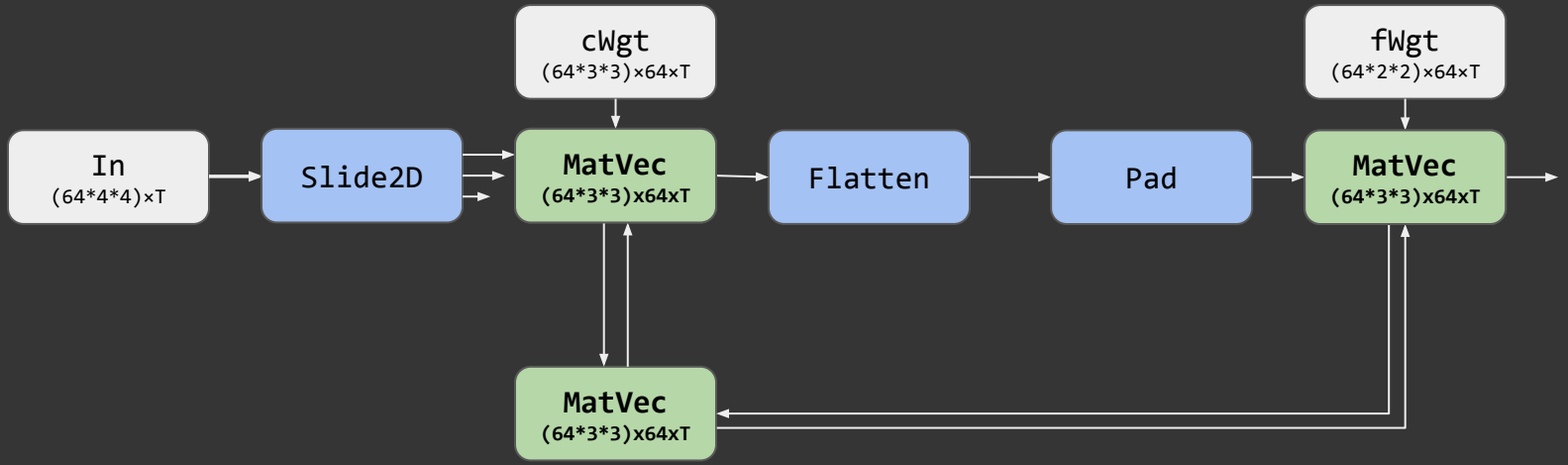


Resource Sharing

Ingredients:

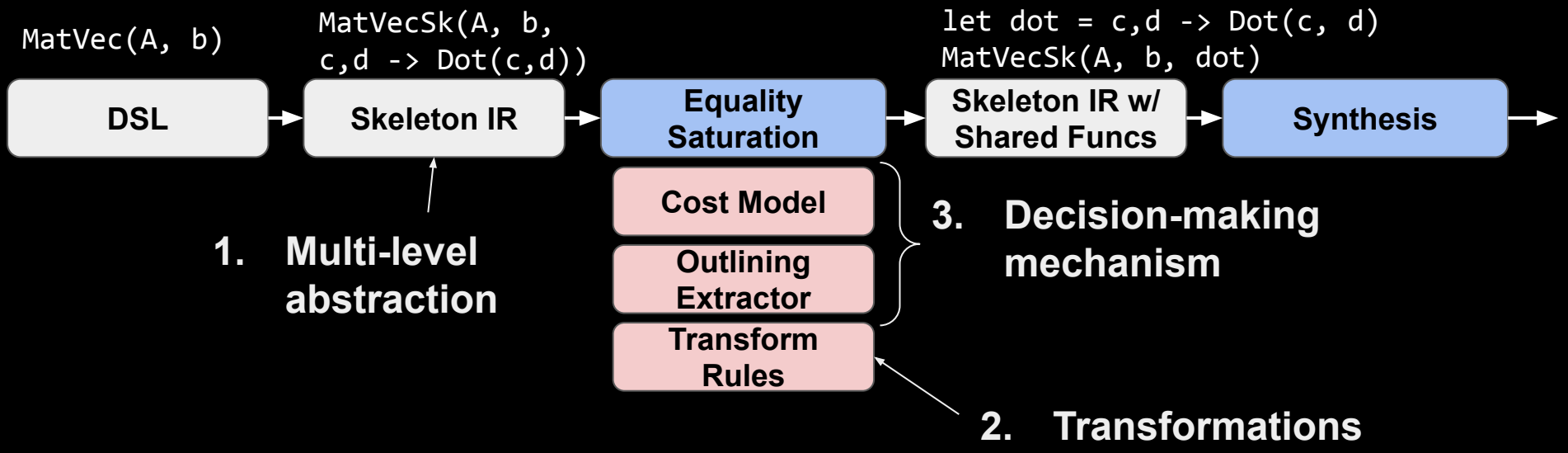
1. Multi-level abstraction
2. Transformations
3. Decision-making mechanism

```
MatVec(fWgt, Flatten(Conv(cWgt, In)))
```



[15]: Tzung-Han Juang et al. Let Coarse-Grained Resources Be Shared: Mapping Entire Neural Networks on FPGAs. **TECS 2023**

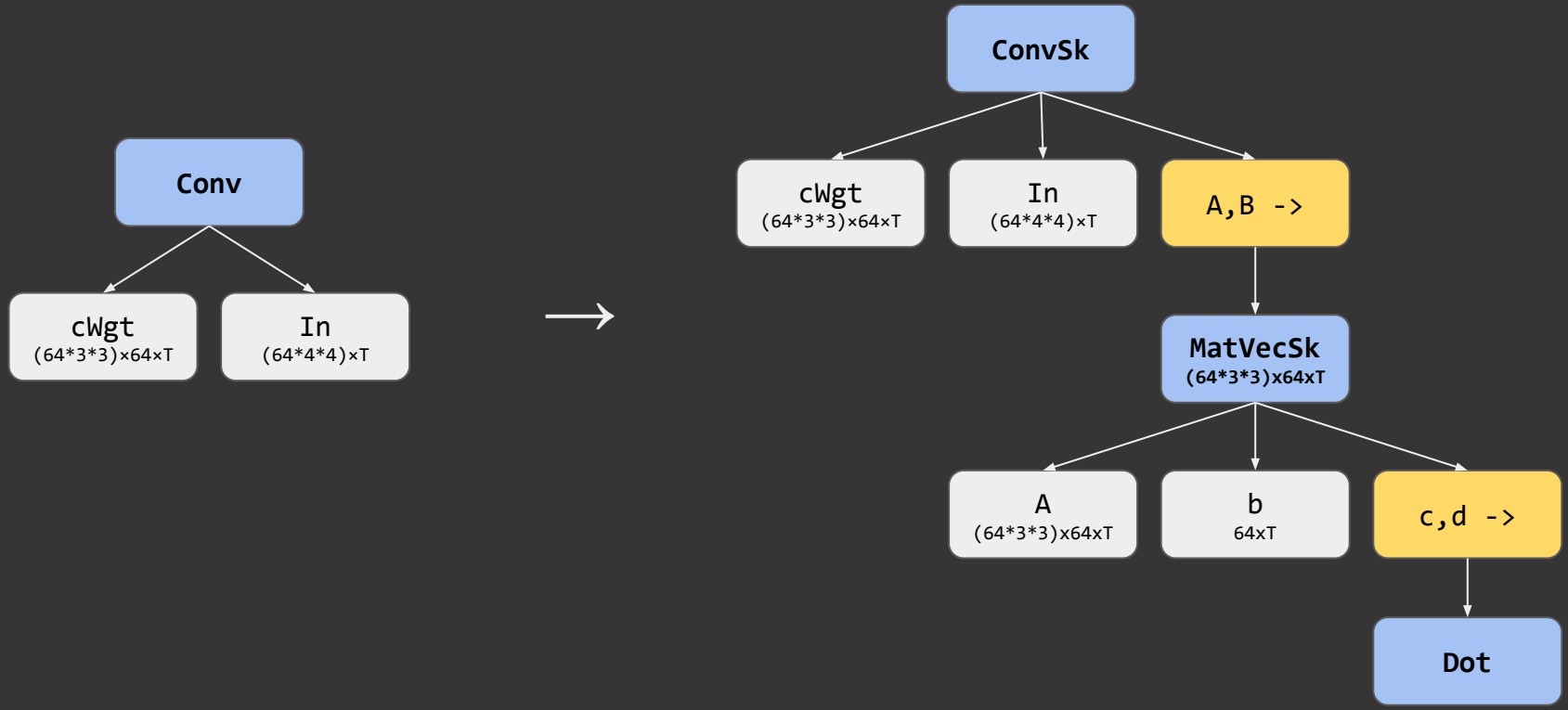
Overview of SkeleShare Approach



Ingredient #1: Skeleton IR

DSL		Skeleton IR
`MatVec(A, b)`	→	`MatVecSk(A, b, c,d -> Dot(c, d))`
`Conv(Wgt, In)`	→	`ConvSk(Wgt, In, A,b -> MatVecSk(A, b, …))`

Skeleton IR


Conv
cWgt
(64*3*3)×64×T
In
(64*4*4)×T
ConvSk
cWgt
(64*3*3)×64×T
In
(64*4*4)×T
A,B ->
MatVecSk
(64*3*3)x64xT
A
(64*3*3)x64xT
b
64xT
c,d ->
Dot

Ingredient #2: Transformations

`Conv`	● Tile
	  ○ Width/height
	  ○ Output channels
	  ○ Input channels
	● Pad width/height

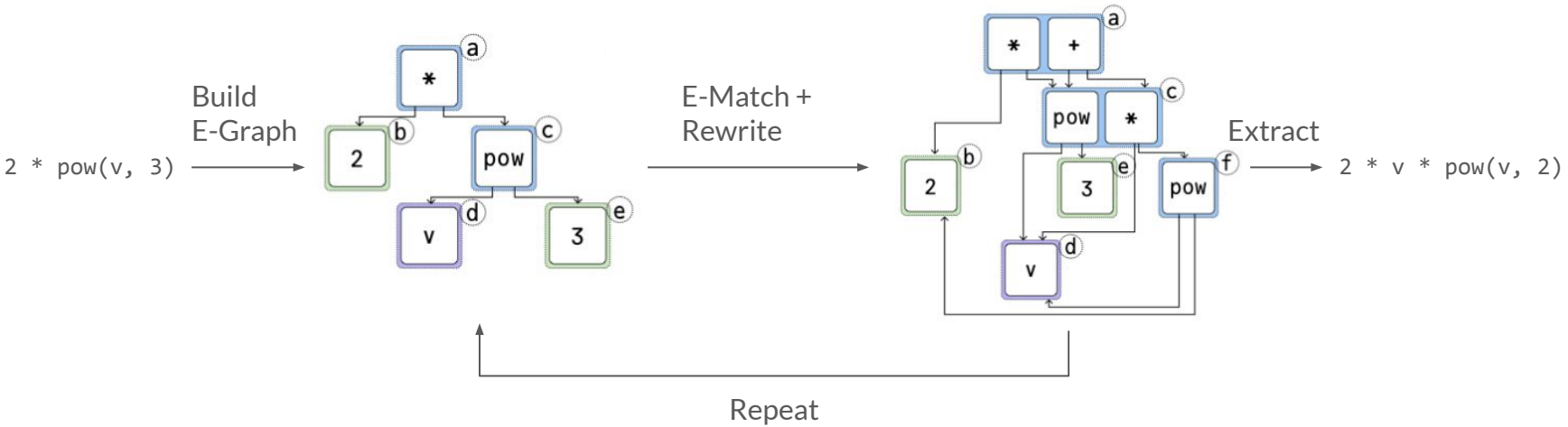
`MatVec`	● Tile ● Pad
`Dot`	● Change parallelism

Transformations

DSL		Skeleton IR
MatVec(A, b)	→	**MatVecSk**(A, b, c,d -> **Dot**(c, d))
	=	**TiledMatVecSk**(A, B, C,D -> **MatVecSk**(C, D, ...))
	=	**PaddedMatVecSk**(A, B, C,D -> **MatVecSk**(C, D, ...))
Conv(Wgt, In)	→	**ConvSk**(Wgt, In, A,B -> **MatVecSk**(A, B, ...))

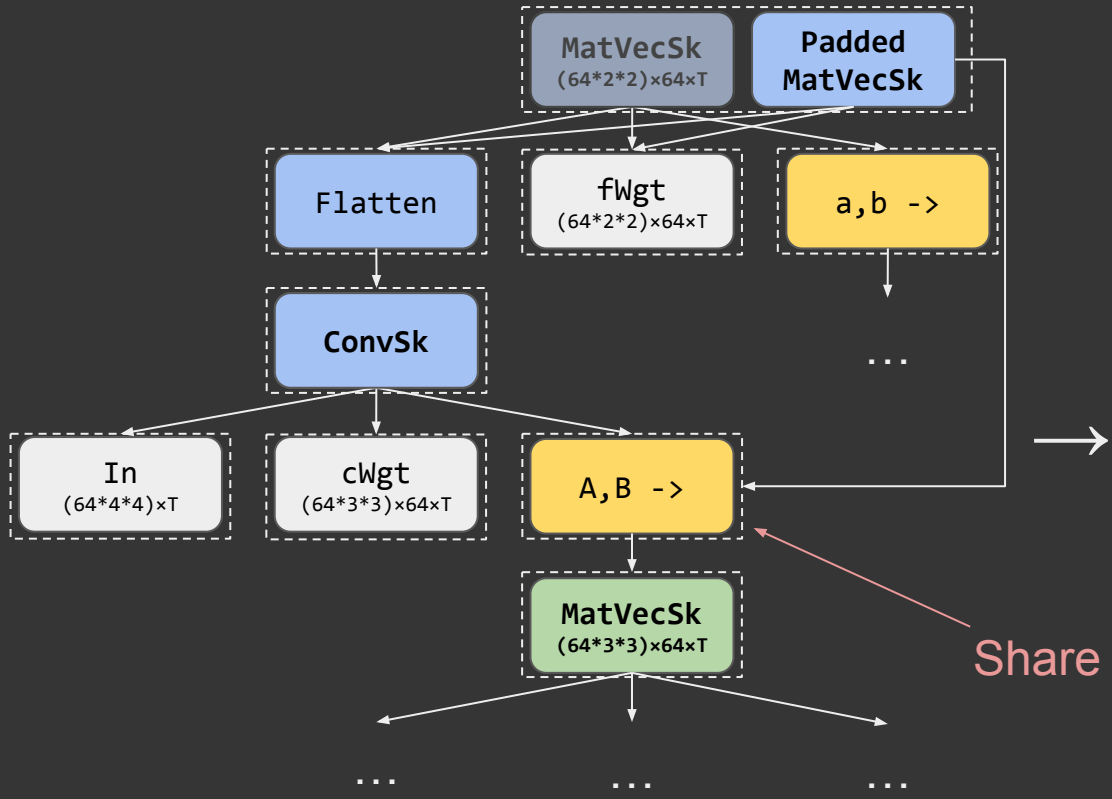
Ingredient #3: Equality Saturation

```
"pow n": pow(x, n) ⇒ x * pow(x, n - 1)
"mul 2": 2 * y     ⇒ y + y
```



Ingredient #3: Equality Saturation

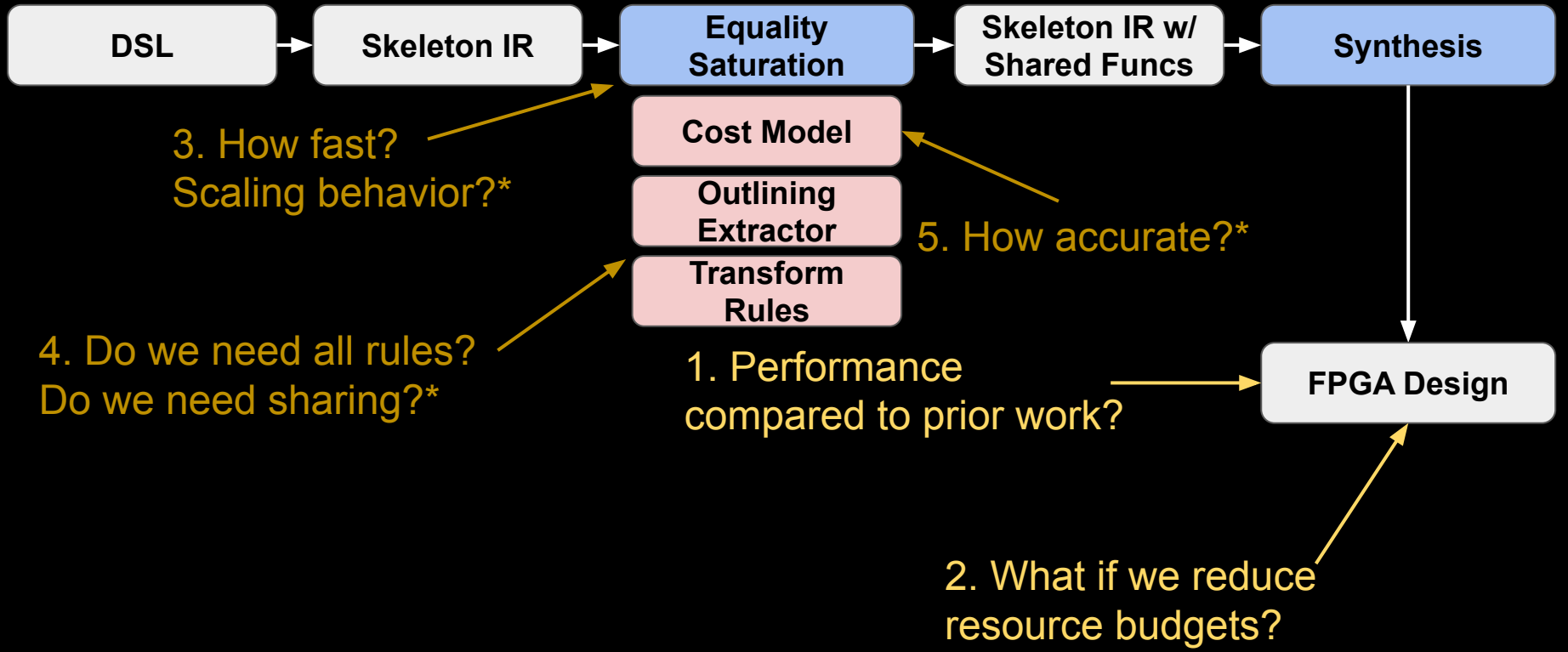
- Extraction
- Construction
- MatVec padding rule



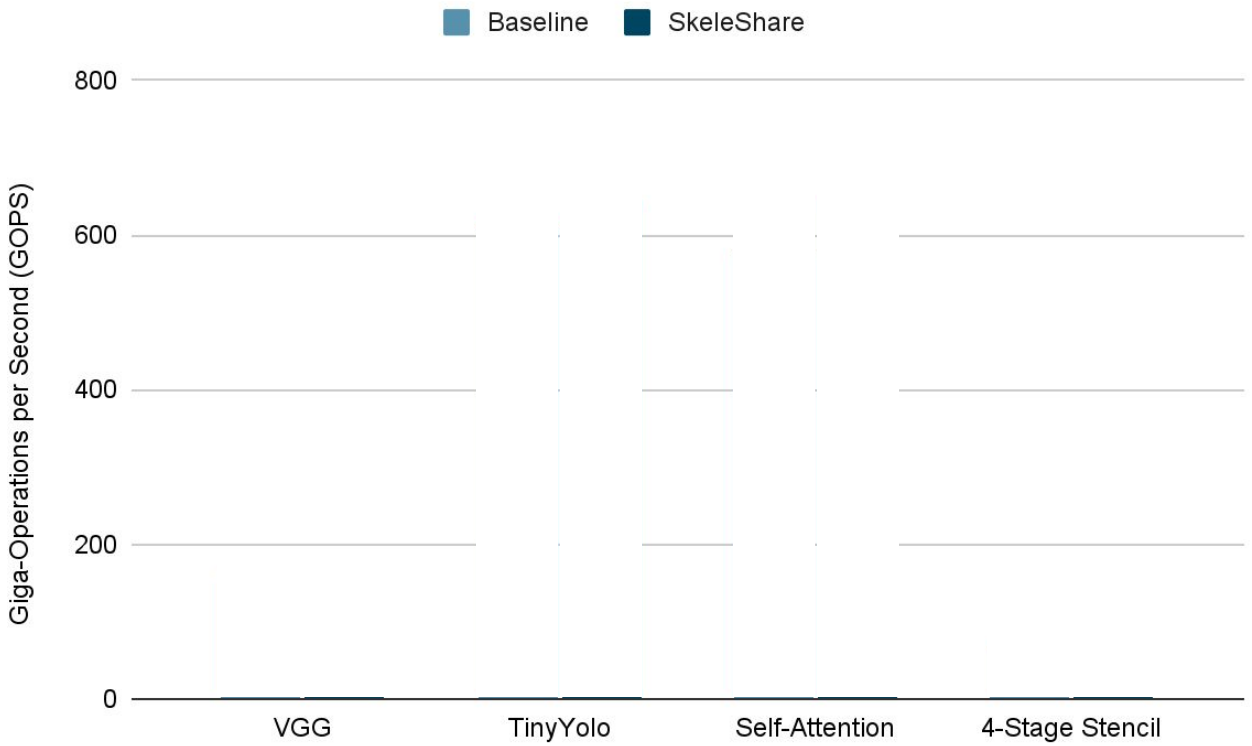

```
let mv = A,B ->
  MatVecSk(64*3*3)x64xT(...)

PaddedMatVecSk(
  fWgt,
  Flatten(ConvSk(
    cWgt, In, mv)),
  mv)
```

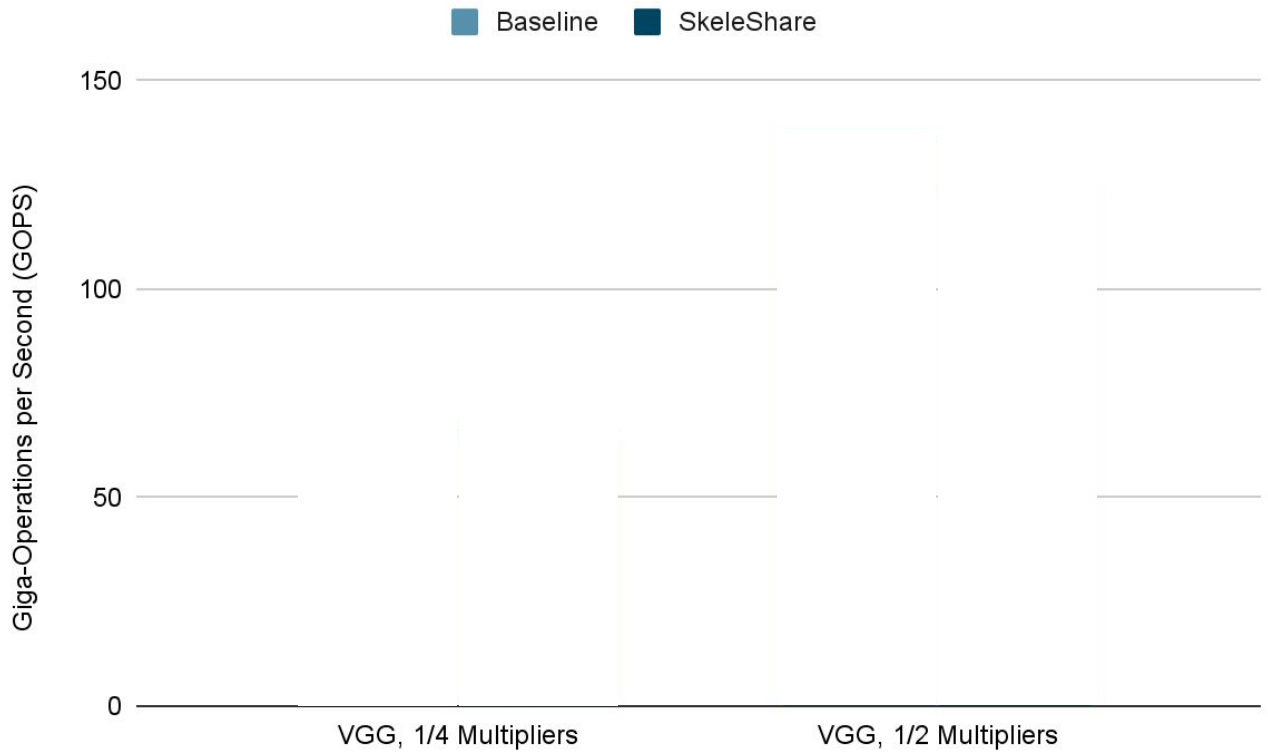
Evaluation


DSL
Skeleton IR
Equality Saturation
Skeleton IR w/ Shared Funcs
Synthesis
Cost Model
Outlining Extractor
Transform Rules
3. How fast?
Scaling behavior?*
5. How accurate?*
4. Do we need all rules?
Do we need sharing?*
1. Performance compared to prior work?
FPGA Design
2. What if we reduce resource budgets?

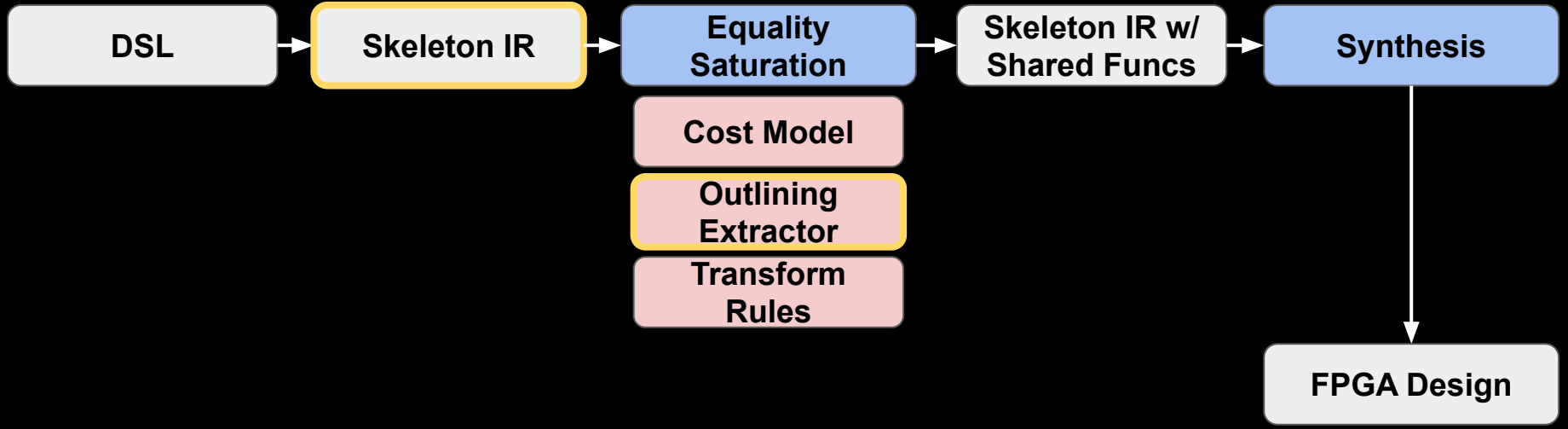
Q1. Comparison to Prior Work



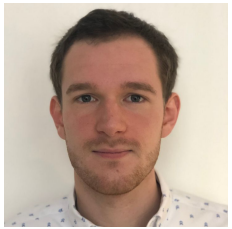
Q2. Smaller Resource Budgets



Conclusion


DSL
Skeleton IR
Equality Saturation
Skeleton IR w/ Shared Funcs
Synthesis
Cost Model
Outlining Extractor
Transform Rules
FPGA Design

SkeleShare: Algorithmic Skeletons and Equality Saturation for Hardware Resource Sharing



Jonathan Van der Cruysse
McGill University
jonathan.vandercruysse
@mail.mcgill.ca

Tzung-Han Juang
McGill University
tzung-han.juang
@mail.mcgill.ca

Shakiba Bolbolian Khah
McGill University
shakiba.bolboliankhah
@mail.mcgill.ca

Christophe Dubach
McGill University & MILA
christophe.dubach
@mcgill.ca

Paper (Jonathan's web site)

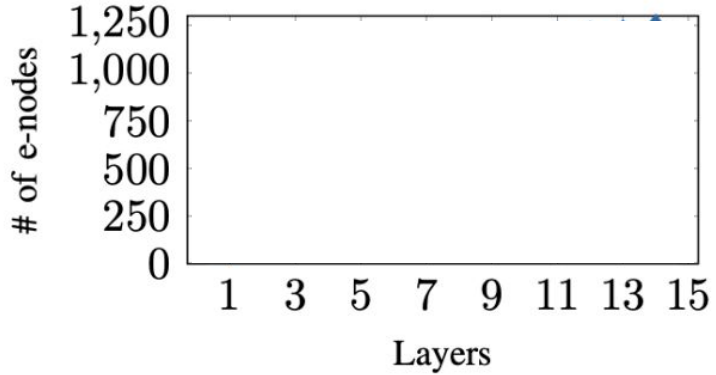
Q1. Comparison to Prior Work

Experiment	Logic	RAM	DSPs	GOPS
1. VGG, SkeleShare	49%	35%	76%	163
3. TinyYolo, SkeleShare	38%	24%	76%	647
6. Self-attention, SkeleShare	35%	29%	67%	648
8. 4-stage stencil, SkeleShare	10%	7%	6%	61

Q2. Smaller Resource Budgets

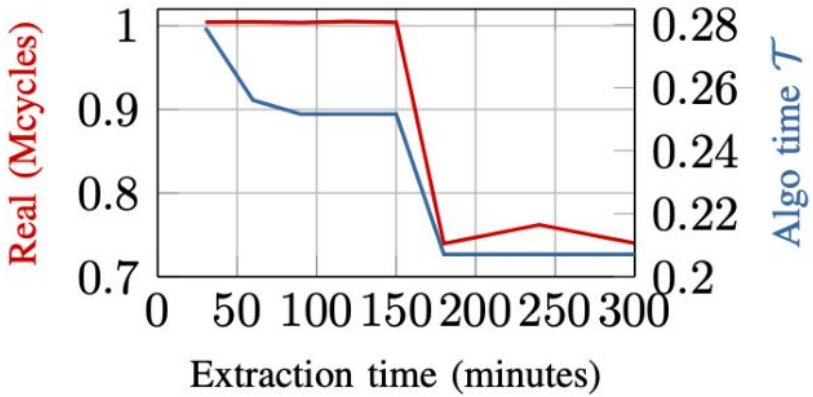
Experiment	Logic	RAM	DSPs	GOPS
15. VGG, SkeleShare, 1/4 DSPs				
17. VGG, SkeleShare, 1/2 DSPs				

Q3. Equality Saturation Speed, Scaling



(a) Number of e-nodes for different numbers of layers.

Q4. Cost Model vs Reality



Q5. Necessity of Rules, Sharing

Experiment	Logic	RAM	DSPs	GOPS
10. VGG, SkeleShare, no sharing	No solution found			
11. VGG, SkeleShare, no padding	No solution found			
12. VGG, SkeleShare, no tiling	No solution found			
13. VGG, baseline, no sharing	Not synthesizable			
14. VGG, SkeleShare, 1 abstr.	45%	35%	72%	125

Figure 1

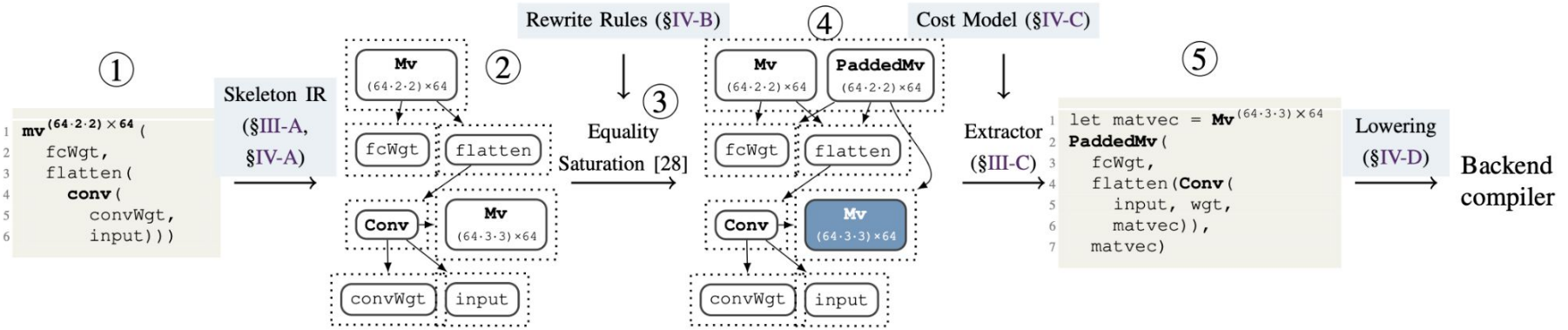


Fig. 1: Overview of SkeleShare, the proposed technique. Program ① is converted to skeleton IR e-graph ②. Equality saturation applies transformation rules ③ to e-graph ②. After rule application, we obtain e-graph ④. The blue-highlighted **Mv** has multiple uses and is hence a candidate for sharing. From e-graph ④, an outlining extractor chooses expression ⑤. Superscripts and e-node annotations denote **Mv** input dimensions. Light blue boxes are domain-specific components.

Figure 2

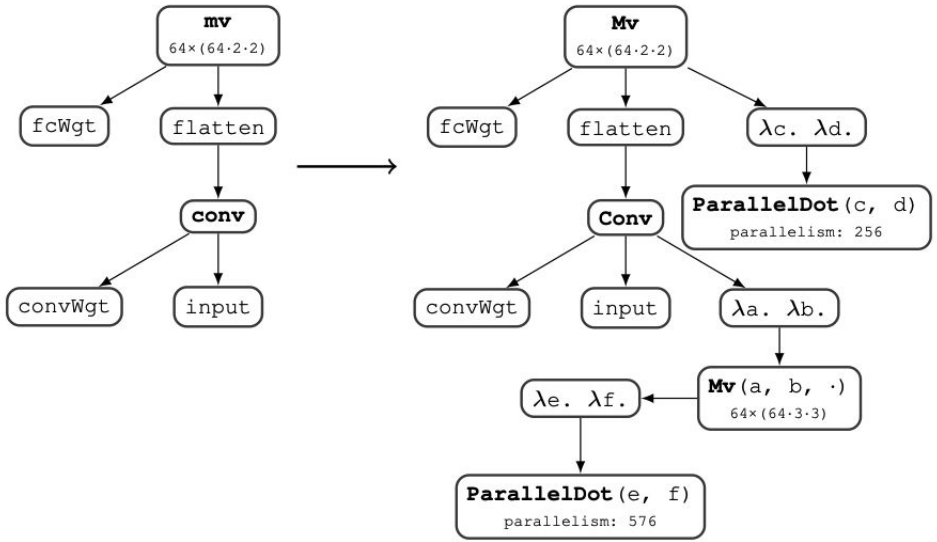


Fig. 2: A tree representation of the running example **mv**(fcWgt, flatten(**conv**(convWgt, input))) on the left, alongside its skeleton IR tree on the right.

Figure 2

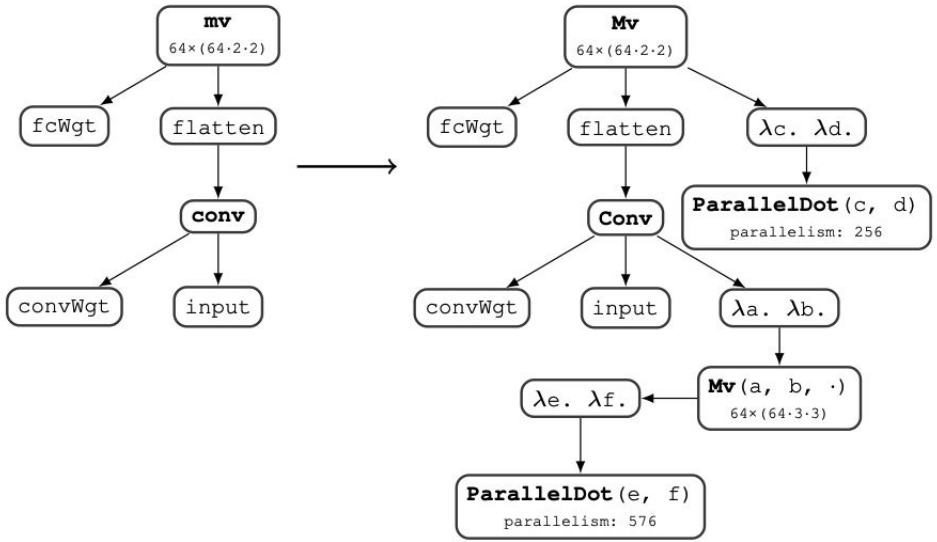


Fig. 2: A tree representation of the running example **mv**(fcWgt, flatten(**conv**(convWgt, input))) on the left, alongside its skeleton IR tree on the right.

Figure 3

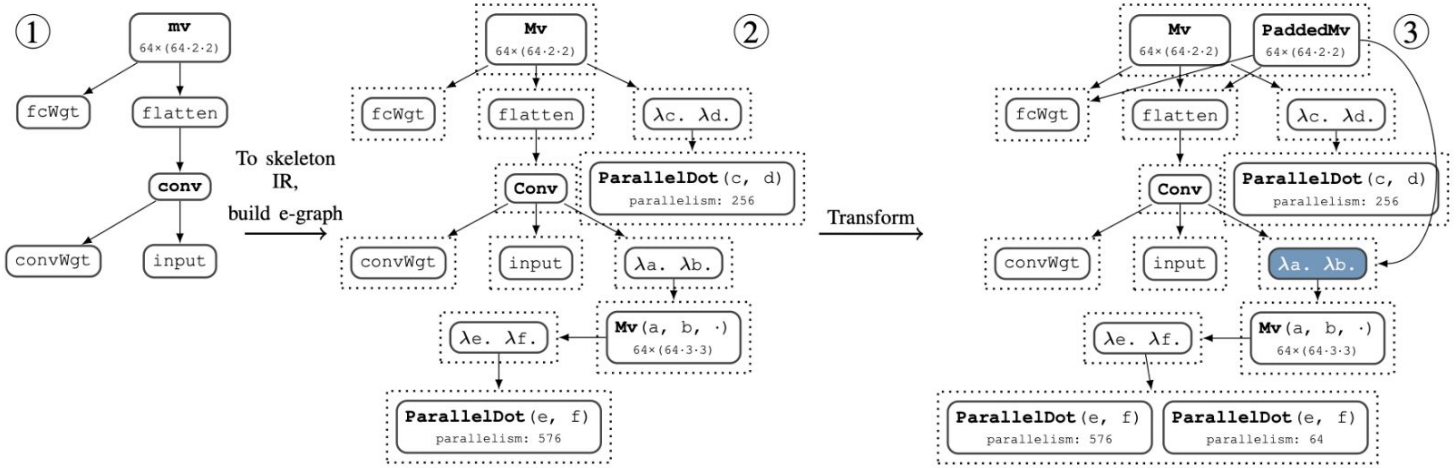


Fig. 3: E-Graph construction and transformation for **mv**(fcWgt, flatten(**conv**(convWgt, input))). Expression tree ① is translated to skeleton IR and encoded as e-graph ②. Rewrite rules expand the graph to ③, introducing a padded **Mv** and multiple **ParallelDot** variants. The top e-class (dotted box) includes both, exposing reuse and performance–cost tradeoffs. Shareable components are in blue; solid boxes are e-nodes, dotted boxes are e-classes.

Figure 4

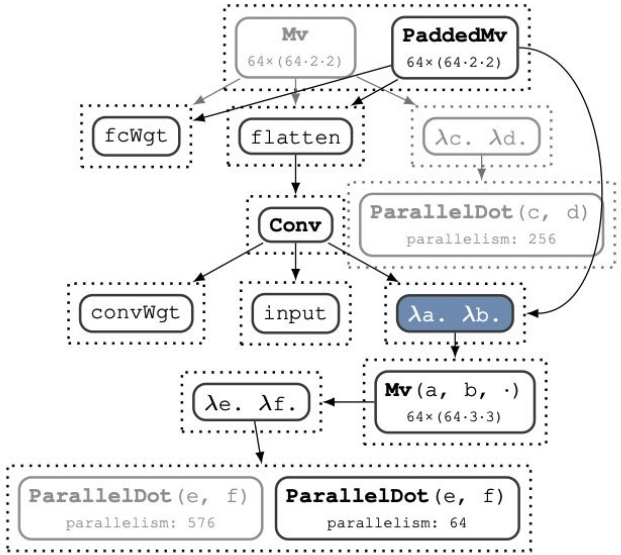


Fig. 4: Example selection and sharing decisions in an e-graph. The sole e-node of a shared e-class is highlighted in blue. Selected e-nodes and e-classes retain their usual color. Non-selected e-nodes and e-classes appear grayed out.

Figure 5

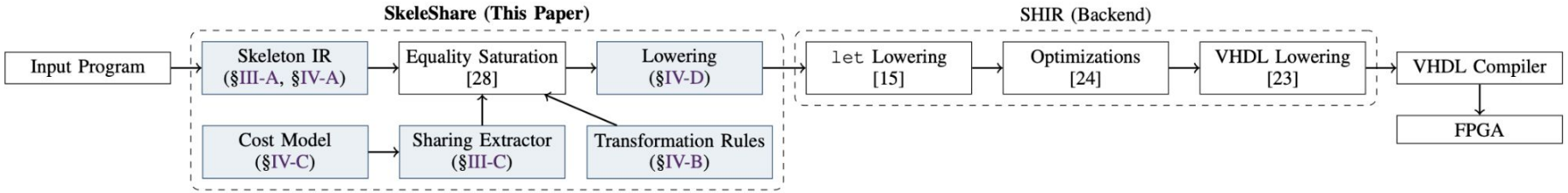


Fig. 5: End-to-end compiler stack integrating SkeleShare with the SHIR pipeline. Blue boxes show SkeleShare's contributions. These produce a skeleton IR that is lowered to SHIR's mid-level primitives and synthesized by a standard FPGA toolchain.

Figure 6

ParallelDot[P](a: M × N × int, b: M × N × int): M × int

Mv(mat: M × N × T, vec: N × T, dotF: *ParallelDotT* [M, N, T, T′]): M × T′

Mm(a: H × K × T, b: W × K × T, mvF: *MvT* [W, K, T, T′]): H × W × T′

Conv(in:H × W × ICH × T, wgt:OCH × KS × KS × ICH × T, mvF:*MvT* [OCH, KS · KS · ICH, T, T′]):(H − KS + 1) × (W − KS + 1) × OCH × T′

Fig. 6: Core skeletons for parallel dot products, matrix-vector products and dot products. By convention, type parameters that can be inferred from arguments are kept implicit. **ParallelDot** is defined over integer inputs, the typical scalar type for FPGAs, while the others are polymorphic. This polymorphism enables flexible reuse, used by partial computation skeletons.

Figure 7

TiledMv(mat: $M \times N \times \text{int}$, vec: $N \times \text{int}$, mvF: $\mathit{MvT}\,[\text{TM}, N, \text{int}, \text{int}]$): $M \times \text{int}$

PaddedMv(mat: $M \times N \times \text{int}$, vec: $N \times \text{int}$, mvF: $\mathit{MvT}\,[M, \text{PN}, \text{int}, \text{int}]$): $M \times \text{int}$

TiledMmW(a: $H \times K \times \text{int}$, b: $W \times K \times \text{int}$, mmF: $\mathit{MmT}\,[\text{TW}, H, K, \text{int}, \text{int}]$): $H \times W \times \text{int}$

TiledMmH(a: $H \times K \times \text{int}$, b: $W \times K \times \text{int}$, mmF: $\mathit{MmT}\,[W, \text{TH}, K, \text{int}, \text{int}]$): $H \times W \times \text{int}$

TiledMmK(a: $H \times K \times \text{int}$, b: $W \times K \times \text{int}$, mmF: $\mathit{MmT}\,[W, H, \text{TK}, \text{int}, \text{int}]$): $H \times W \times \text{int}$

TiledConvHW(in: $H \times W \times \text{ICH} \times \text{int}$, wgt: $\text{OCH} \times \text{KS} \times \text{KS} \times \text{ICH} \times \text{int}$, convF: $\mathit{ConvT}\,[\text{TW}, \text{TH}, \text{ICH}, \text{OCH}, \text{KS}, \text{int}, \text{int}]$):
$(H - \text{KS} + 1) \times (W - \text{KS} + 1) \times \text{OCH} \times \text{int}$

TiledConvICH(in: $H \times W \times \text{ICH} \times \text{int}$, wgt: $\text{OCH} \times \text{KS} \times \text{KS} \times \text{ICH} \times \text{int}$, convF: $\mathit{ConvT}\,[W, H, \text{TICH}, \text{OCH}, \text{KS}, \text{int}, \text{int}]$):
$(H - \text{KS} + 1) \times (W - \text{KS} + 1) \times \text{OCH} \times \text{int}$

TiledConvOCH(in: $H \times W \times \text{ICH} \times \text{int}$, wgt: $\text{OCH} \times \text{KS} \times \text{KS} \times \text{ICH} \times \text{int}$, convF: $\mathit{ConvT}\,[W, H, \text{ICH}, \text{TOCH}, \text{KS}, \text{int}, \text{int}]$):
$(H - \text{KS} + 1) \times (W - \text{KS} + 1) \times \text{OCH} \times \text{int}$

PaddedConvHW(in: $H \times W \times \text{ICH} \times \text{int}$, wgt: $\text{OCH} \times \text{KS} \times \text{KS} \times \text{ICH} \times \text{int}$, convF: $\mathit{ConvT}\,[\text{PW}, \text{PH}, \text{ICH}, \text{OCH}, \text{KS}, \text{int}, \text{int}]$):
$(H - \text{KS} + 1) \times (W - \text{KS} + 1) \times \text{OCH} \times \text{int}$

PaddedConvICH(in: $H \times W \times \text{ICH} \times \text{int}$, wgt: $\text{OCH} \times \text{KS} \times \text{KS} \times \text{ICH} \times \text{int}$, convF: $\mathit{ConvT}\,[W, H, \text{PICH}, \text{OCH}, \text{KS}, \text{int}, \text{int}]$):
$(H - \text{KS} + 1) \times (W - \text{KS} + 1) \times \text{OCH} \times \text{int}$

Fig. 7: Data-transformed skeleton definitions. Padding, tiling and transposition type parameters are highlighted in yellow.

Figure 8

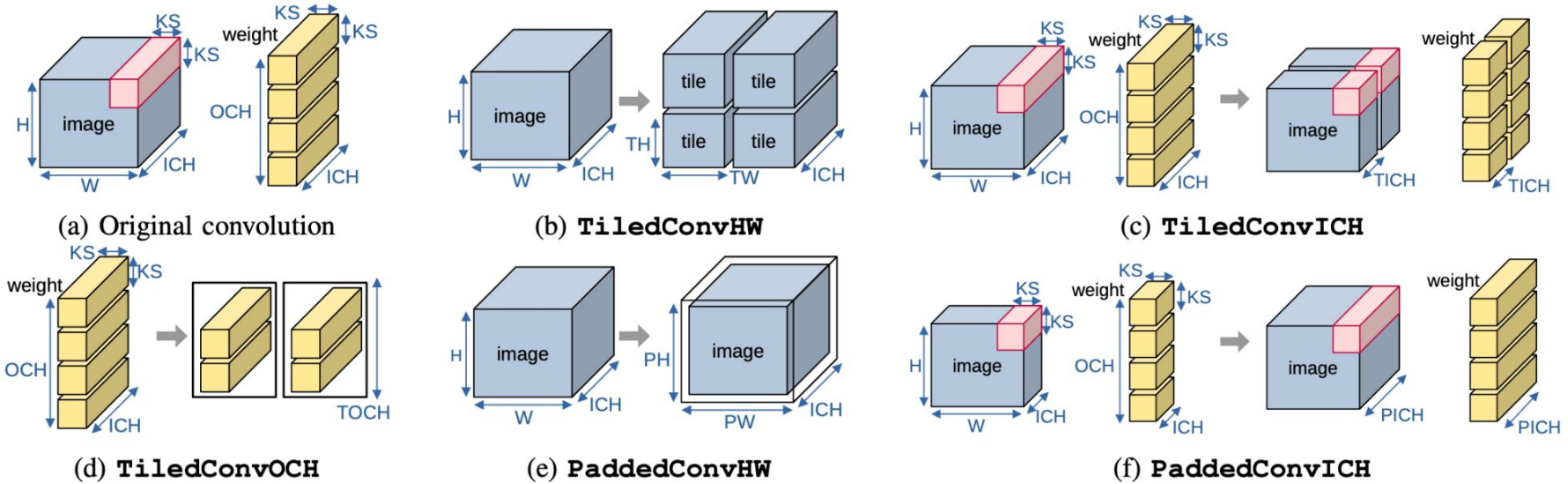


Fig. 8: Visualization of tiling and padding opportunities on convolutions. Tiling and padding can be applied to any of the four dimensions of a convolution. Data-transformed skeletons cover tiling on all four dimensions, splitting up larger convolutions into smaller tiles. The skeletons also describe padding on the height, width, and input channel dimensions, extending input sizes to share hardware with an existing larger convolution.

Figure 9

```
MergePartialSum(elements: M × C × int): M × int
ParallelPartialDot[P,C](a: M × N × int, b: M × N × int): M × C × int
ParallelConv(in: H × W × ICH × T, wgt: OCH × KS × KS × ICH × T, mvF: MvT[OCH, C · KS · KS · ICH, T, C × T′]):
    (H − KS + 1) × (W − KS + 1) × OCH × T′
```

Fig. 9: Partial dot product and parallel convolution skeletons.

Figure 10

Conv(in: $H \times W \times ICH \times \text{int}$, wgt: $OCH \times KS \times KS \times ICH \times \text{int}$, _) $\rightarrow$ **TiledConvHW**(in, wgt, **cConvF** $[TH, TW, ICH, OCH, KS]$)
where $H = 0 \bmod TH$ and $W = 0 \bmod TW$ (R-TILE-CONV-HW)

Conv(in: $H \times W \times ICH \times \text{int}$, wgt: $OCH \times KS \times KS \times ICH \times \text{int}$, _) $\rightarrow$ **TiledConvICH**(in, wgt, **cConvF** $[H, W, TICH, OCH, KS]$)
where $ICH = 0 \bmod TICH$ (R-TILE-CONV-ICH)

Conv(in: $H \times W \times ICH \times \text{int}$, wgt: $OCH \times KS \times KS \times ICH \times \text{int}$, _) $\rightarrow$ **TiledConvOCH**(in, wgt, **cConvF** $[H, W, ICH, TOCH, KS]$)
where $OCH = 0 \bmod TICH$ (R-TILE-CONV-OCH)

Conv(in: $H \times W \times ICH \times \text{int}$, wgt: $OCH \times KS \times KS \times ICH \times \text{int}$, _) $\rightarrow$ **PaddedConvHW**(in, wgt, **cConvF** $[PH, PW, ICH, OCH, KS]$)
where $PH > H$ and $PW > W$ (R-PAD-CONV-HW)

Mm(a: $H \times K \times \text{int}$, b: $W \times K \times \text{int}$, _) $\rightarrow$ **TiledMmW**(a, b, **cMmF** $[TW, H, K]$) where $W = 0 \bmod TW$ (R-TILE-MMW)

Mm(a: $H \times K \times \text{int}$, b: $W \times K \times \text{int}$, _) $\rightarrow$ **TiledMmH**(a, b, **cMmF** $[W, TH, K]$) where $H = 0 \bmod TH$ (R-TILE-MMH)

Mm(a: $H \times K \times \text{int}$, b: $W \times K \times \text{int}$, _) $\rightarrow$ **TiledMmK**(a, b, **cMmF** $[W, H, TK]$) where $K = 0 \bmod TK$ (R-TILE-MMK)

Mv(mat: $M \times N \times \text{int}$, vec: $N \times \text{int}$, _) $\rightarrow$ **TiledMv**(mat, vec, **cMvF** $[M, TN]$) where $N = 0 \bmod TN$ (R-TILE-MV)

Mv(mat: $M \times N \times \text{int}$, vec: $N \times \text{int}$, _) $\rightarrow$ **PaddedMv**(mat, vec, **cMvF** $[M, PN]$) where $PN > N$ (R-PAD-MV)

where **cMvF** $[M, N] = \lambda$mat: $M \times N \times \text{int}$. λvec: $N \times \text{int}$. **Mv**(mat, vec, λa. λb. **ParallelDot** $[M]$(a, b))

cMmF $[W, H, K] = \lambda$a: $H \times K \times \text{int}$. λb: $W \times K \times \text{int}$. **Mm**(a, b, **cMvT**$[W, K]$)

cConvF $[H, W, ICH, OCH, KS] = \lambda$in: $H \times W \times ICH \times \text{int}$. λwgt: $OCH \times KS \times KS \times ICH \times \text{int}$.
Conv(in, wgt, **cMvF** $[OCH, KS \cdot KS \cdot ICH, \text{int}, \text{int}]$)

Fig. 10: Data transformation rules.

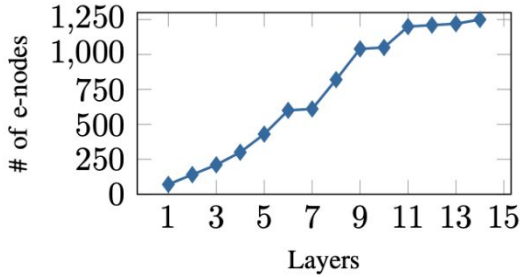
Figure 11. Partial Parallel Convolution Padding Rule

$$\begin{aligned}&\textbf{Conv}(\texttt{in: } H \times W \times ICH \times \texttt{int},\ \texttt{wgt: } OCH \times KS \times KS \times ICH \times \texttt{int},\ \_)\\ &\quad\rightarrow \textbf{PaddedConvICH}(\texttt{in},\ \texttt{wgt},\ \lambda \texttt{pi: } H \times W \times PICH \times \texttt{int}.\ \lambda \texttt{pw: } OCH \times KS \times KS \times PICH \times \texttt{int}.\\ &\qquad \textbf{ParallelConv}(\texttt{pi},\ \texttt{pw},\ \textbf{pMvF}\,[C, OCH, C \times PICH])),\end{aligned} \tag{R-PAD-PAR-CONV}$$

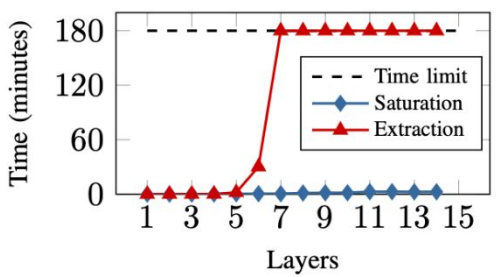
where $\textbf{pMvF}\,[C, M, N] = \lambda \texttt{mat: } M \times N \times \texttt{int}.\ \lambda \texttt{vec: } N \times \texttt{int}.\ \textbf{Mv}(\texttt{mat},\ \texttt{vec},\ \lambda \texttt{a}.\ \lambda \texttt{b}.\ \textbf{ParallelPartialDot}\,[C, M](\texttt{a},\ \texttt{b}))$

Fig. 11: Partial parallel convolution padding rule.

Figures 12, 13. Scaling Behavior, Cost Model Predictions



(a) Number of e-nodes for different numbers of layers.



(b) Compilation time for different numbers of layers.

Fig. 12: Scaling behavior of SkeleShare on increasingly large slices of VGG-CIFAR. x-axis indicates the number of layers (*e.g.*, $x = 5$ uses first five layers).

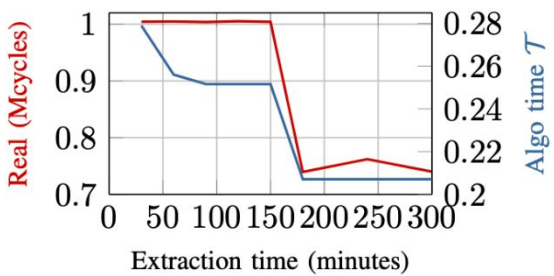


Fig. 13: Cost model predictions and measured hardware runtime for VGG under different solver timeouts.

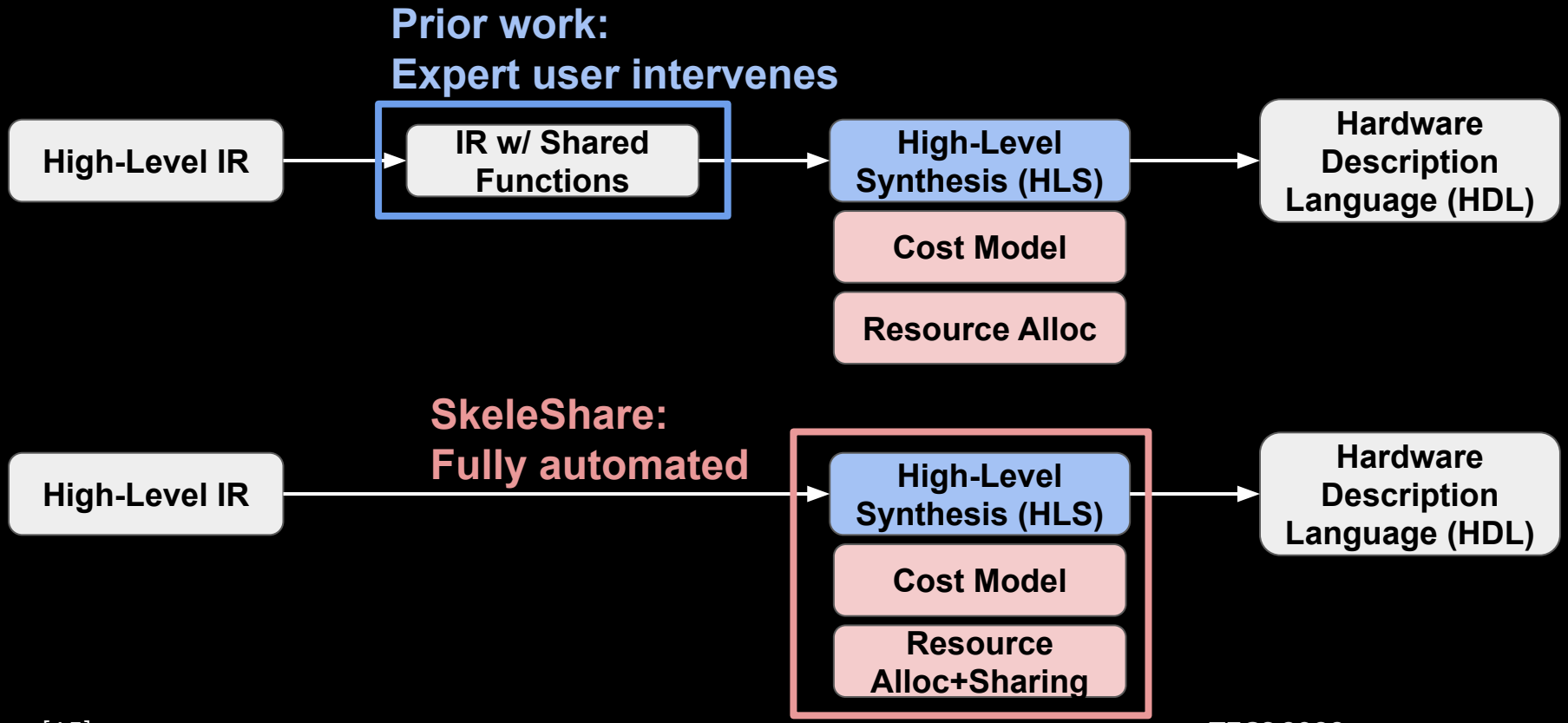
Table III. Overview of Experiments

TABLE III: Performance evaluation of SkeleShare on VGG and TinyYolo. Logic, RAM, and DSPs show resource use. Throughput metric: GOPS (Giga-Operations Per Second). Prior work does not report logic/RAM use for TinyYolo [15].

Experiment	Logic	RAM	DSPs	GOPS
1. VGG, SkeleShare	49%	35%	76%	163
2. VGG, [15]	48%	33%	76%	166
3. TinyYolo, SkeleShare	38%	24%	76%	647
4. TinyYolo, [15]	N/A	N/A	76%	608
5. TinyYolo, [33]	N/A	N/A	58%	500
6. Self-attention, SkeleShare	35%	29%	67%	648
7. Self-attention, [18][1]	10%	7%	14%	583
8. 4-stage stencil, SkeleShare	10%	7%	6%	61
9. 4-stage stencil, baseline	11%	7%	25%	63
10. VGG, SkeleShare, no sharing	No solution found			
11. VGG, SkeleShare, no padding	No solution found			
12. VGG, SkeleShare, no tiling	No solution found			
13. VGG, baseline, no sharing	Not synthesizable			
14. VGG, SkeleShare, 1 abstr.	45%	35%	72%	125
15. VGG, SkeleShare, 1/4 DSPs	42%	35%	19%	65
16. VGG, [15], 1/4 DSPs	40%	33%	19%	69
17. VGG, SkeleShare, 1/2 DSPs	40%	35%	38%	123
18. VGG, [15], 1/2 DSPs	42%	33%	38%	136

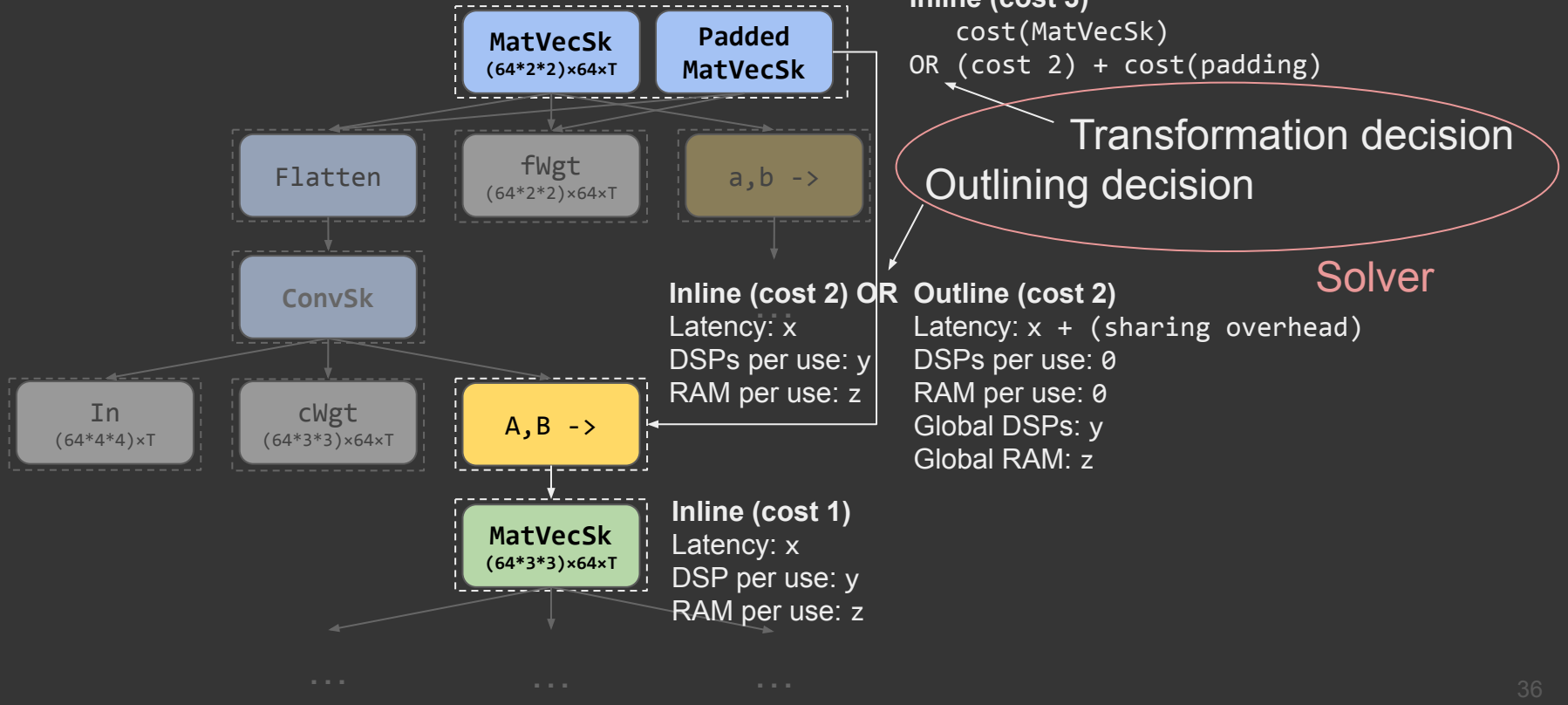
[1] Evaluated on Xilinx Virtex Ultrascale+ with more hardware resources than Intel Arria 10. Both self-attention designs use 1024 DSPs.

From Semi-Automatic to Automatic Partitioning



[15]: Tzung-Han Juang et al. Let Coarse-Grained Resources Be Shared: Mapping Entire Neural Networks on FPGAs. **TECS 2023**

Equality Saturation: Extraction and Outlining



MatVec Skeleton

```
MatVec(A, b) = map(
  (a, b’) -> Dot(a, b’),
  zip(A, repeat(B)))
```

```
MatVecSk(A, b, f) = map(
  f,
  zip(A, repeat(B)))
```