Hardware-aware numerical data formats for DNN acceleration

Anastasia Volkova
with Bastien Barbe, Romain Bouarah, Florent de Dinechin
Inria, INSA Lyon

RAIM and JMM days
November 4, 2025

Motivation and approach

Motivation and approach

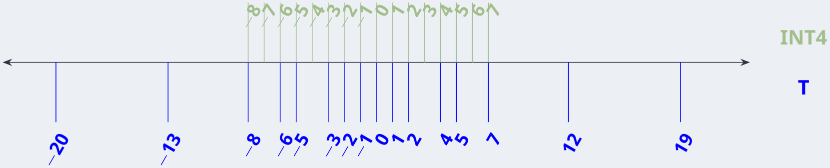
- DNN acceleration is ~~what gets you funding~~ an active research area
- Wild wild west of numerical data formats (BF16, FP8 ocp/hybrid, FP6, FP4, ...)
- Need to reduce the algorithm-hardware gap

Our approach to building a custom inference accelerator:

- Look into efficient arithmetic operators in hardware
- If needed devise new HW-friendly formats
- Quantize/tune DNNs to fit the hardware
- Let the DNNs learn the best formats

Hardware-aware number formats

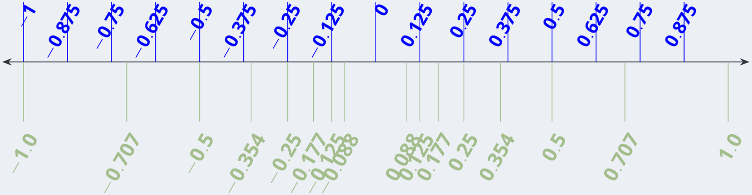
Shift-and-Add friendly



- Multiplication by each target constant can be done with only 2 adders
- Each multiplication is configured with only 4 bits

Hardware-aware number formats

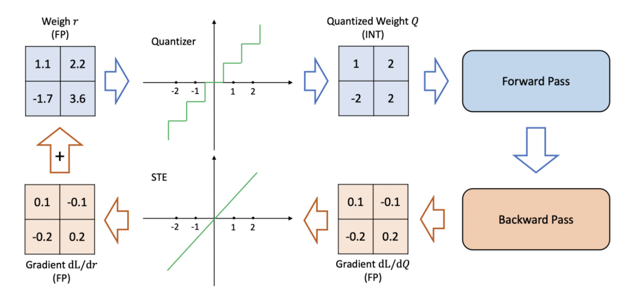
Logarithmic Number System



- $LNS(b, m, \ell) = \{(-1)^s \cdot b^{-L_X} \mid s \in \{0, 1\}, L_X \in ufix(m, \ell)\}$
- Cheap multiplication but costly addition
- LNS is a candidate for representing normal distributions

Adapting DNNs to HW-friendly formats

- **HAtorch**: hardware-aware quantization-aware training
- Yet another training framework... but
  - > any data format is a set of points and a rounding function
  - > exploits autograd as much as it can
  - > no hidden (floating-point) scaling factors
  - > control over weights/activations/functions



Spoiler-alert: it seems to work

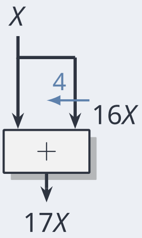
- Includes quantization information during retraining
- Full accuracy (ResNet-56 on CIFAR-100)
- Individual RSCMs are competitive with INT4 on FPGAs
- LNS-neuron implementation is pending

Weight format	Top-1 Accuracy	Top-5 Accuracy
FP32 baseline	75.09 %	93.05 %
INT4	74.87 %	93.53 %
RSCM4	75.19 %	93.42 %
$LNSU(3.46, 1, -4)$	75.16 %	93.82 %

Shift-and-Add aware format

Single Constant Multiplier (SCM)

- Compute $X \times C$ using only shifts and additions/subtractions
- C known at design time, X variable



$$17X = 16X + X = (X \ll 4) + X$$

Figure: An SCM where $C = 17$.

Reconfigurable Single Constant Multiplier (RSCM)

- Compute $X \times T_i$ where $T_i \in T$, constants known at design time
- i is index of the constant chosen at run time (hence reconfigurable)

$i_1 i_0$	T_i
01	$-3X = X - 4X = (X \ll 2) + X$
00	$5X = X + 4X = (X \ll 2) + X$
11	$-15X = X - 16X = (X \ll 4) + X$
10	$17X = X + 16X = (X \ll 4) + X$

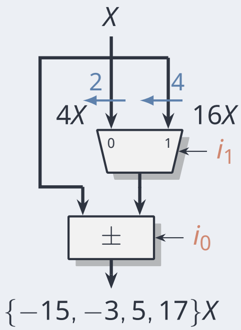


Figure: An RSCM where $T = \{-15, -3, 5, 17\}$.

State of the art

Objective

Given a set of constants T, find the RSCM that minimizes a given cost function

Main contributions

- Optimal algorithm in its model; prior work used heuristics[1] or less expressive models[2]
- Increase $\#T$ beyond prior limit of 20
- Bit-level cost accounting for full- and half-adders
- Application to CNN inference

[1] **tummeltshammer2007time**, **tummeltshammer2007time**, **tummeltshammer2007time**.
[2] **eleftheriadis2023optimal**, **eleftheriadis2023optimal**, **eleftheriadis2023optimal**.

Solver overview

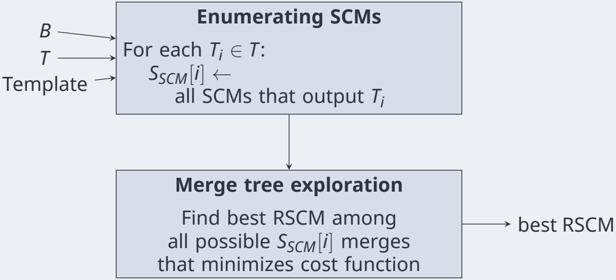


Figure: Overview of the 2-step algorithm

- Inputs:
  - B Constant bit-width
  - T Set of constants
  - Template: common structure for all generated SCMs, independent of T_i
- From the same template, generate all SCMs for each $T_i \in T$
- Build RSCMs by merging SCMs

Templates: three adders

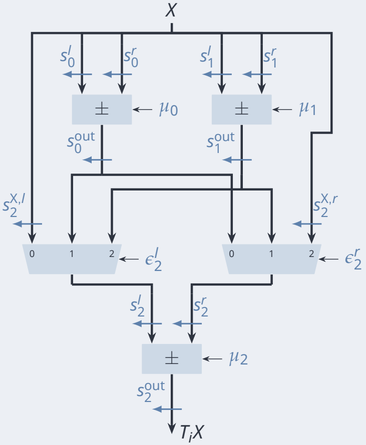


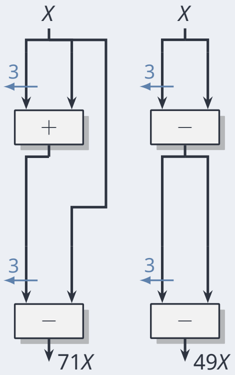
Figure: 3 adders on 2 layers template.

Table: CP variables for adder i of depth > 0

Variable	Description
$s_i^l \in [\![0, B-1]\!]$	shift on the left input
$s_i^r \in [\![0, B-1]\!]$	shift on the right input
$s_i^{out} \in [\![0, B-1]\!]$	shift on the output
$\mu_i \in \{0, 1\}$	addition or subtraction
$\epsilon_i^l \in [\![0, \alpha_i]\!]$	left input MUX selector
$\epsilon_i^r \in [\![0, \alpha_i]\!]$	right input MUX selector
$s_i^{X,l} \subset [\![0, B-1]\!]$	shift on X if input to the left
$s_i^{X,r} \subset [\![0, B-1]\!]$	shift on X if input to the right

Building an RSCM: the merging process

- Build RSCMs by merging SCMs
- Insert multiplexers wherever variable assignments differ



(a) (b)

Building an RSCM: the merging process

- Build RSCMs by merging SCMs
- Insert multiplexers wherever variable assignments differ

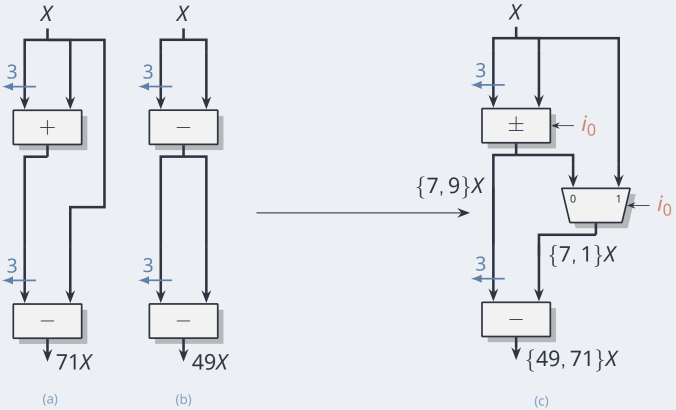


Figure: Merging two SCMs (a) and (b) to obtain an RSCM (c) for $T = \{49, 71\}$ with one configuration bit i_0.

Building an RSCM: the merging process

- Build RSCMs by merging SCMs
- Insert multiplexers wherever variable assignments differ

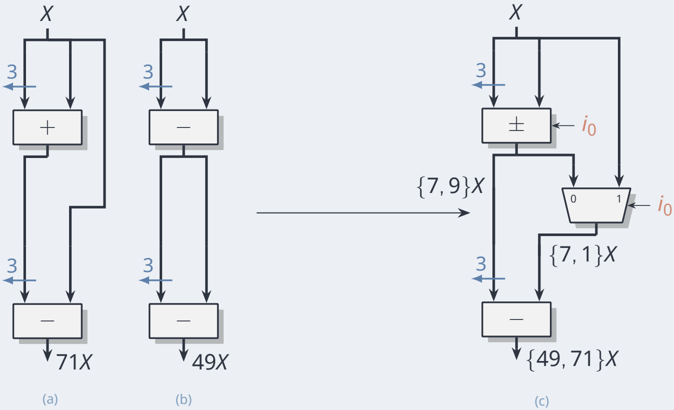


Figure: Merging two SCMs (a) and (b) to obtain an RSCM (c) for $T = \{49, 71\}$ with one configuration bit i_0.

- In practice, SCMs and RSCMs are stored as bitsets
  > Merging reduces to OR-ing their bitsets (constant time)
- Multiple SCM solutions for $T_i = 71$ and $T_j = 49$ imply multiple RSCMs for $T = \{71, 49\}$
  > Merges to explore: $\prod_{i=0}^{n-1} \#S_{SCM}[i]$

Cost functions

- Prior work used MUX2 count as the cost
- But solutions with equal MUX2 count can have different area

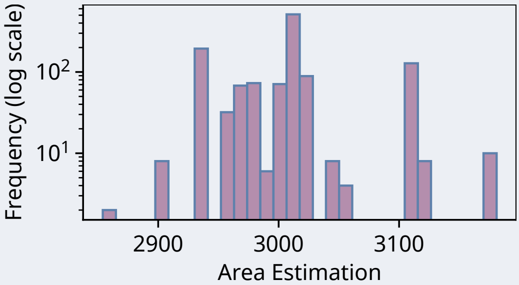


Figure: Area cost distribution for the 1213 RSCM architectures implementing the same constants set within optimal MUX2 count.

Cost functions

- Prior work used MUX2 count as the cost
- But solutions with equal MUX2 count can have different area

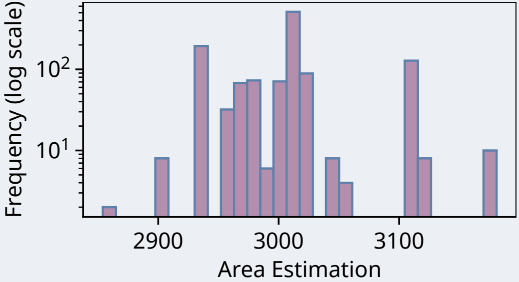


Figure: Area cost distribution for the 1213 RSCM architectures implementing the same constants set within optimal MUX2 count.

- Use finer-grained area estimates accounting for full- and half-adders
- Tailor the cost to a specific platform or technology node

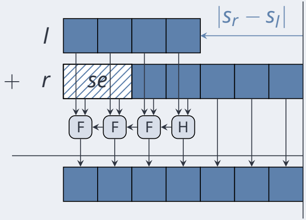


Figure: Counting one-bit adders for the adder case, *se* denotes the sign extension.

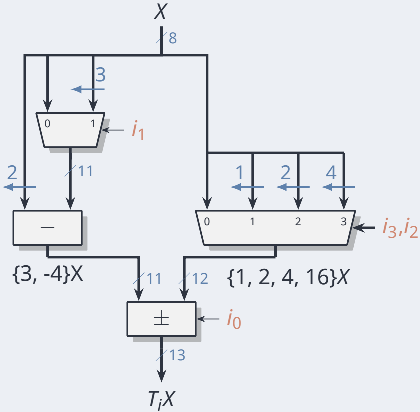
Comparison with previous works

- Lowest #MUX2 in all tested cases, up to 55% reduction vs DAG Fusion
- Scales to 256 constants (prior work was limited to 20)
- MUX2 cost is not optimal under fine-grained cost, but up to 43× faster
  - > Mitigate with hybrid cost: warm start with coarse, then refine with fine-grained
- Limitations: up to 3 adders and < 12-bit constants

A toy example: 6-bit weights dynamic encoded in 4 bits

Motivation

Replacing multipliers with RSCMs for efficient machine learning inference.



i_3	i_2	i_1	i_0	T_i
0	0	0	0	4
0	0	0	1	2
0	0	1	0	-3
0	0	1	1	-5
0	1	0	0	5
0	1	0	1	1
0	1	1	0	-2
0	1	1	1	-6

i_3	i_2	i_1	i_0	T_i
1	0	0	0	7
1	0	0	1	-1
1	0	1	0	0
1	0	1	1	-8
1	1	0	0	19
1	1	0	1	-13
1	1	1	0	12
1	1	1	1	-20

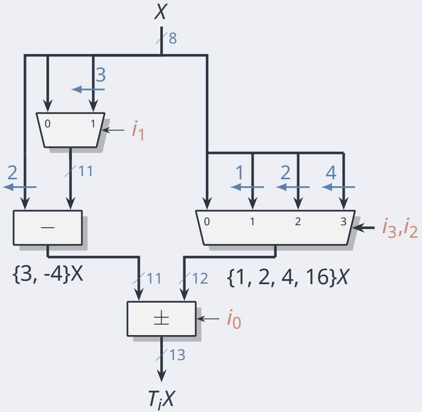
Memory efficiency

This RSCM allows to compute constants in a 6-bit range using only 4 bits to store them !

A toy example: 6-bit weights dynamic encoded in 4 bits

Motivation

Replacing multipliers with RSCMs for efficient machine learning inference.



i_3	i_2	i_1	i_0	T_i	i_3	i_2	i_1	i_0	T_i
0	0	0	0	4	1	0	0	0	7
0	0	0	1	2	1	0	0	1	-1
0	0	1	0	-3	1	0	1	0	0
0	0	1	1	-5	1	0	1	1	-8
0	1	0	0	5	1	1	0	0	19
0	1	0	1	1	1	1	0	1	-13
0	1	1	0	-2	1	1	1	0	12
0	1	1	1	-6	1	1	1	1	-20

Memory efficiency

This RSCM allows to compute constants in a 6-bit range using only 4 bits to store them !

$$T_i \in \{-20, -13, -8, -6, -5, -3, -2, -1, 0, 1, 2, 4, 5, 7, 12, 19\}$$

Comparison against INT4 and INT6: Hardware Metrics

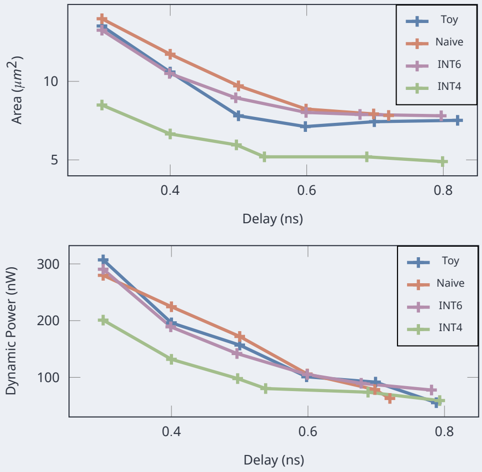


Figure: ASIC performance metrics (TSMC 4*nm* node).

Table: Synthesis on FPGA (AMD Kintex 7)

	latency	area
Toy	2.611ns	31 LUT
INT4	3.133ns	29 LUT
INT6	3.182ns	48 LUT
Naive	3.451ns	54 LUT

- FPGA: toy offers best latency for marginally more area
- ASIC: toy competitive around 0.4–0.7*ns* delay

LNS-neuron

LNS

$$LNS(b, m, \ell) = \left\{ (-1)^s \cdot b^{-L_X} \mid s \in \{0, 1\}, L_X \in ufix(m, \ell) \right\}$$

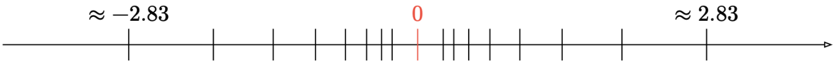


Figure 3: LNS(2, 1, −1)

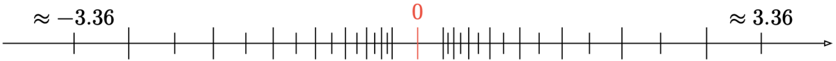


Figure 4: LNS(2, 1, −2)

Figure 5: LNS(2, 1, −3)

LNS

$$LNS(b, m, \ell) = \{(-1)^s \cdot b^{-L_X} \mid s \in \{0,1\}, L_X \in ufix(m, \ell)\}$$

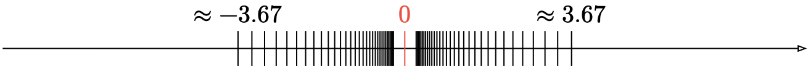


Figure 6: LNS(2, 1, −3)

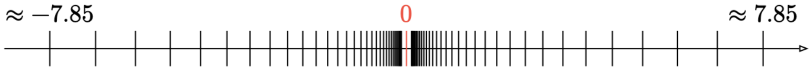


Figure 7: LNS(3, 1, −3)

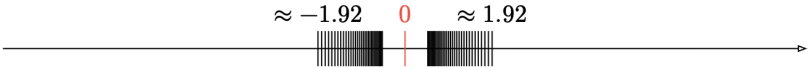


Figure 8: LNS$(\sqrt{2}, 1, -3)$

LNS in hardware

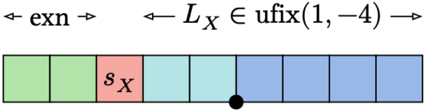


Figure 12: Hardware representation of $X \in \text{LNS}(b, m, l)$

Let X be the number whose hardware representation is given in the figure above:

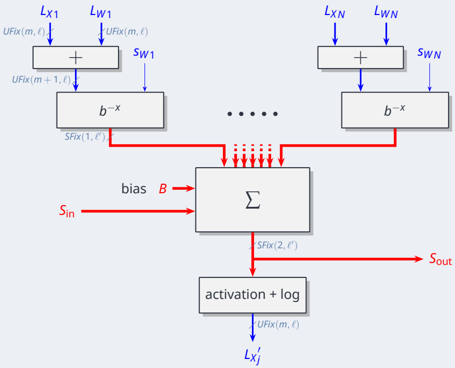
- If exn = 00 then $X = (-1)^{s_X} \cdot b^{L_X}$
- If exn = 01 then $X = 0$
- If exn = 10 then $X = (-1)^{s_X} \times \infty$
- If exn = 11 then X is not a number

Operations:

- $X \times Y \to L_{X \times Y} = L_X + L_Y$
- $X + Y \to L_{X+Y} = L_X + \log_b\left(\left|1 + (-1)^{s_Y - s_X} \cdot b^{L_Y - L_X}\right|\right)$

LNS neuron

- The LNS Neuron [Christ'22]: scalar product, activation function and conversions back to log
- Choosing the "good" base for the table s.t. zero is represented -> even better to "learn" it!
- SGD converegence suffers under rounding to logarithmic domain
- Challenge to find tradeoff between LNS and conversion accuracy



Conclusion and on-going work

Conclusion and on-going work

- Hardware-aware algorithms to bridge the efficiency gap
- QAT is a generic approach to adapt DNNs to arithmetic but does not solve the higher-level problem of the model adequacy itself
- Currently work on improving the HAtorch tool and a framework for generic architecture generation
- Looking into mixed/low-precision training and its convergence

Comparison with previous works

Table: Average optimal number of MUX2 in 2-adder RSCMs for $\#T$ constants of B bits

B	5			6			8			10			12		
$\#T$	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
DAG Fusion (run)	4.53	8.48	13.3	5.27	9.34	13.97	6.93	11.51	16.06	7.71	12.74	18.35	8.33	13.89	19.75
TMCCM (data from paper[4])	3.41	6.11	/	3.74	6.65	/	/	/	/	/	/	/	/	/	/
This work, $\pm$ is free	2.30	3.24	4.13	2.51	3.92	5.14	3.31	5.21	6.93	3.92	6.03	8.10	4.22	6.75	9.12
This work, $\pm$ costs a MUX2	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Savings compared to DAG Fusion	26%	44%	55%	33%	41%	50%	35%	40%	45%	34%	39%	45%	35%	38%	44%

[4] **eleftheriadis2023optimal**, **eleftheriadis2023optimal**, **eleftheriadis2023optimal**.

Comparison with previous works

Table: Average optimal number of MUX2 in 2-adder RSCMs for $\#T$ constants of B bits

B	5			6			8			10			12		
$\#T$	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
DAG Fusion (run)	4.53	8.48	13.3	5.27	9.34	13.97	6.93	11.51	16.06	7.71	12.74	18.35	8.33	13.89	19.75
TMCCM (data from paper[4])	3.41	6.11	/	3.74	6.65	/	/	/	/	/	/	/	/	/	/
This work, $\pm$ is free	2.30	3.24	4.13	2.51	3.92	5.14	3.31	5.21	6.93	3.92	6.03	8.10	4.22	6.75	9.12
This work, $\pm$ costs a MUX2	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Savings compared to DAG Fusion	26%	44%	55%	33%	41%	50%	35%	40%	45%	34%	39%	45%	35%	38%	44%

- Lowest #MUX2 in all tested cases, up to 55% vs DAG Fusion

[4] **eleftheriadis2023optimal**, **eleftheriadis2023optimal**, **eleftheriadis2023optimal**.

Comparison with previous works

Table: Average optimal number of MUX2 in 2-adder RSCMs for $\#T$ constants of B bits

B	5			6			8			10			12		
$\#T$	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
DAG Fusion (run)	4.53	8.48	13.3	5.27	9.34	13.97	6.93	11.51	16.06	7.71	12.74	18.35	8.33	13.89	19.75
TMCCM (data from paper[4])	3.41	6.11	/	3.74	6.65	/	/	/	/	/	/	/	/	/	/
This work, $\pm$ is free	2.30	3.24	4.13	2.51	3.92	5.14	3.31	5.21	6.93	3.92	6.03	8.10	4.22	6.75	9.12
This work, $\pm$ costs a MUX2	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Savings compared to DAG Fusion	26%	44%	55%	33%	41%	50%	35%	40%	45%	34%	39%	45%	35%	38%	44%

- Lowest #MUX2 in all tested cases, up to 55% vs DAG Fusion
- Scales to 256 constants (prior work limited to 20)

Table: #MUX2 cost for larger $\#T$ (timeout 5 minutes)

$\#T$ \ B	8	10	12
32	10.53	12.06	13.28
64	12.66	15.42	17.84
128	14.14	18.44	21.62
256	–	21.08	26.17

[4] **eleftheriadis2023optimal**, **eleftheriadis2023optimal**, **eleftheriadis2023optimal**.

Cost functions and runtime considerations

Table: #MUX2, area costs and average solving time by constant set for the three cost functions

	B	5			6			8			10			12		
	#*T*	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
#MUX2 cost	Coarse-grained	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
	Fine-grained	3.44	5.15	6.46	3.80	6.01	7.57	4.96	7.67	9.75	5.70	8.61	11.10	6.02	9.71	12.16
	Hybrid	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Area cost	Coarse-grained	2303	2977	3429	2601	3667	3920	3272	4213	4894	3772	4753	5780	4137	5546	6754
	Fine-grained	2095	2652	3067	2347	3041	3521	2954	3785	4411	3325	4282	5113	3634	4842	5828
	Hybrid	2102	2686	3115	2381	3106	3595	3007	3894	4538	3439	4425	5303	3732	5124	6099
Run-time (s)	Coarse-grained	0.29	0.48	1.27	0.33	0.86	2.25	0.69	1.78	3.86	1.87	4.00	11.03	3.93	9.71	27.62
	Fine-grained	0.31	0.58	5.14	0.36	1.12	18.58	0.72	2.36	66.54	2.03	8.78	469.35	4.40	20.01	990.50
	Hybrid	0.30	0.54	6.32	0.35	1.04	13.46	0.70	2.11	65.17	1.95	4.84	65.08	4.13	12.33	123.82

Cost functions and runtime considerations

Table: #MUX2, area costs and average solving time by constant set for the three cost functions

	B	5			6			8			10			12		
	#T	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
#MUX2 cost	Coarse-grained	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
	Fine-grained	3.44	5.15	6.46	3.80	6.01	7.57	4.96	7.67	9.75	5.70	8.61	11.10	6.02	9.71	12.16
	Hybrid	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Area cost	Coarse-grained	2303	2977	3429	2601	3667	3920	3272	4213	4894	3772	4753	5780	4137	5546	6754
	Fine-grained	2095	2652	3067	2347	3041	3521	2954	3785	4411	3325	4282	5113	3634	4842	5828
	Hybrid	2102	2686	3115	2381	3106	3595	3007	3894	4538	3439	4425	5303	3732	5124	6099
Run-time (s)	Coarse-grained	0.29	0.48	1.27	0.33	0.86	2.25	0.69	1.78	3.86	1.87	4.00	11.03	3.93	9.71	27.62
	Fine-grained	0.31	0.58	5.14	0.36	1.12	18.58	0.72	2.36	66.54	2.03	8.78	469.35	4.40	20.01	990.50
	Hybrid	0.30	0.54	6.32	0.35	1.04	13.46	0.70	2.11	65.17	1.95	4.84	65.08	4.13	12.33	123.82

- MUX2 cost is not optimal under fine-grained cost, but up to 43× faster

Cost functions and runtime considerations

Table: #MUX2, area costs and average solving time by constant set for the three cost functions

	B	5			6			8			10			12		
	#T	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
#MUX2 cost	Coarse-grained	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
	Fine-grained	3.44	5.15	6.46	3.80	6.01	7.57	4.96	7.67	9.75	5.70	8.61	11.10	6.02	9.71	12.16
	Hybrid	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Area cost	Coarse-grained	2303	2977	3429	2601	3667	3920	3272	4213	4894	3772	4753	5780	4137	5546	6754
	Fine-grained	2095	2652	3067	2347	3041	3521	2954	3785	4411	3325	4282	5113	3634	4842	5828
	Hybrid	2102	2686	3115	2381	3106	3595	3007	3894	4538	3439	4425	5303	3732	5124	6099
Run-time (s)	Coarse-grained	0.29	0.48	1.27	0.33	0.86	2.25	0.69	1.78	3.86	1.87	4.00	11.03	3.93	9.71	27.62
	Fine-grained	0.31	0.58	5.14	0.36	1.12	18.58	0.72	2.36	66.54	2.03	8.78	469.35	4.40	20.01	990.50
	Hybrid	0.30	0.54	6.32	0.35	1.04	13.46	0.70	2.11	65.17	1.95	4.84	65.08	4.13	12.33	123.82

- MUX2 cost is not optimal under fine-grained cost, but up to 43× faster
- Mitigate with hybrid cost: warm start with coarse, then refine with fine-grained

Cost functions and runtime considerations

Table: #MUX2, area costs and average solving time by constant set for the three cost functions

	B	5			6			8			10			12		
	$\#T$	4	8	16	4	8	16	4	8	16	4	8	16	4	8	16
#MUX2 cost	Coarse-grained	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
	Fine-grained	3.44	5.15	6.46	3.80	6.01	7.57	4.96	7.67	9.75	5.70	8.61	11.10	6.02	9.71	12.16
	Hybrid	3.33	4.79	6.01	3.55	5.48	6.95	4.53	6.93	8.84	5.13	7.77	10.08	5.44	8.56	11.10
Area cost	Coarse-grained	2303	2977	3429	2601	3667	3920	3272	4213	4894	3772	4753	5780	4137	5546	6754
	Fine-grained	2095	2652	3067	2347	3041	3521	2954	3785	4411	3325	4282	5113	3634	4842	5828
	Hybrid	2102	2686	3115	2381	3106	3595	3007	3894	4538	3439	4425	5303	3732	5124	6099
Run-time (s)	Coarse-grained	0.29	0.48	1.27	0.33	0.86	2.25	0.69	1.78	3.86	1.87	4.00	11.03	3.93	9.71	27.62
	Fine-grained	0.31	0.58	5.14	0.36	1.12	18.58	0.72	2.36	66.54	2.03	8.78	469.35	4.40	20.01	990.50
	Hybrid	0.30	0.54	6.32	0.35	1.04	13.46	0.70	2.11	65.17	1.95	4.84	65.08	4.13	12.33	123.82

- MUX2 cost is not optimal under fine-grained cost, but up to 43× faster
- Mitigate with hybrid cost: warm start with coarse, then refine with fine-grained
  - > 7.2× faster than fine-grained; area gap drops from 11% to 2.9% ($\#T = 16$, $B = 10$)