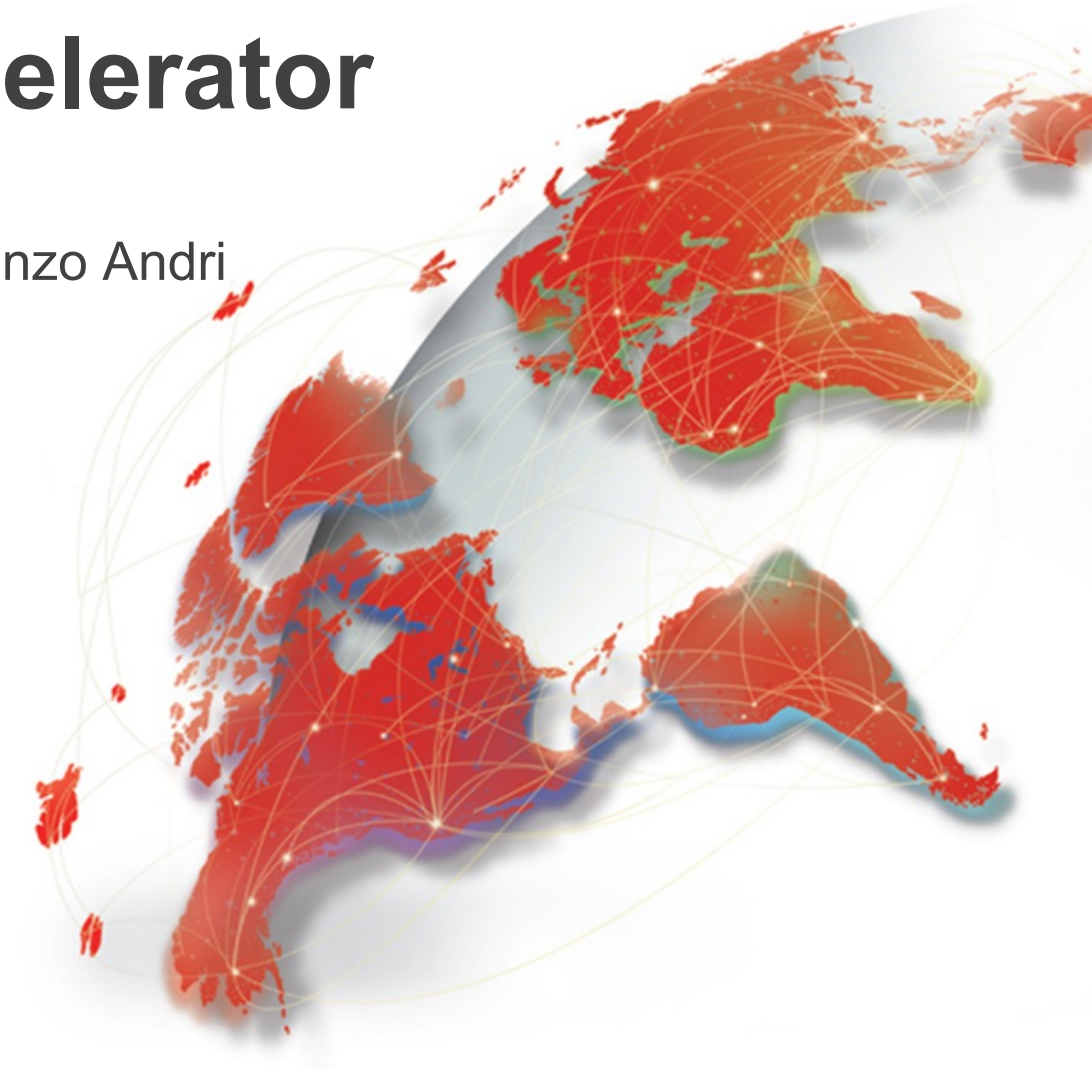
HUAWEI | ZURICH RESEARCH CENTER | COMPUTING SYSTEMS LABORATORY

MA Project: MADDNESS Accelerator

Jannis Schönleber, Matteo Perotti, Lukas Cavigelli, Renzo Andri



Meeting Midway (Week 11)

- **Outline**
  - **Introduction and Overview**
  - **Results**
    - Resnet-50
    - DS-CNN KWS
    - LeViT SOTA transformer classifier
  - **Hardware**
    - Encoder (Layout)
    - Decoder (Layout, area)
    - Scale-out
  - **Conclusion**

MADDness: Method

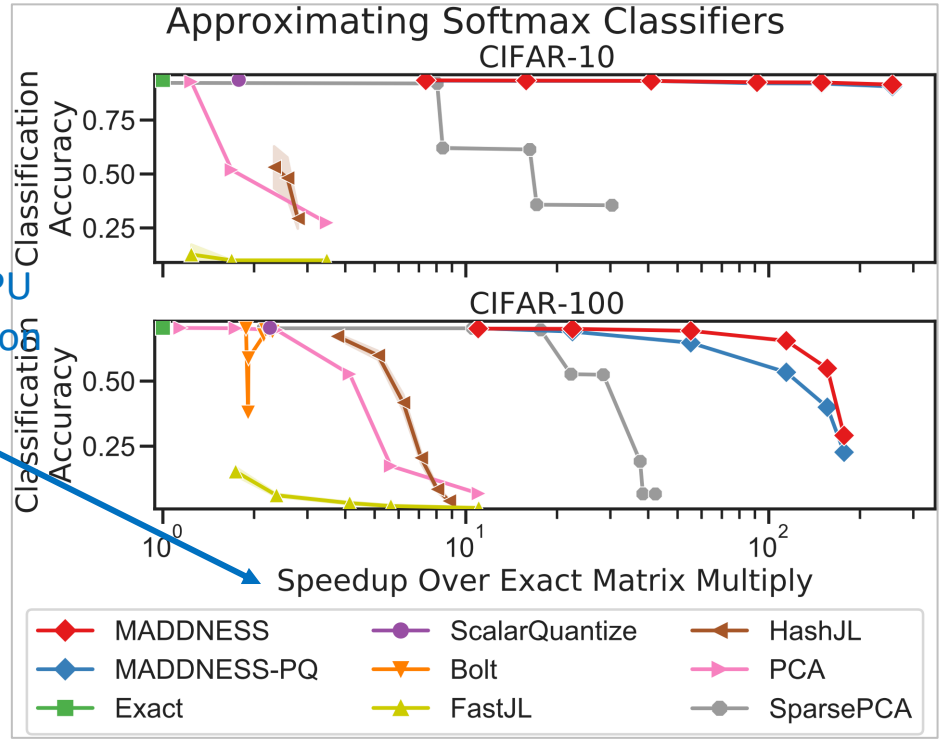
- **What does MADDness solve?**
  - $A \in \mathbb{R}^{N \times D}, B \in \mathbb{R}^{D \times M}, \; N \gg D \geq M$
  - Given compute time budget τ,

    find $enc(.)$, $h(.)$, $dec(.)$, α, β, such that

    $\left\| \alpha \, dec\big(enc(A), h(B)\big) + \beta - AB \right\|_F < \varepsilon(\tau) \|AB\|_F$
- **MADDNESS:**
  - makes use of B matrix being fixed ("weights")
  - input for inference:
    - (1) matrix A,
    - (2) tensor of look-up tables T computed from B
  - $(AB)_{n,m} \approx \alpha \, dec\big(enc(A), h(B)\big)_{n,m} + \beta = \alpha \sum_{c=1}^{C} L_{c,k,m} + \beta$

    with $k = g^{(c)}(a_n)$

based on 1-thread CPU implementation



Algorithm 1 MADDNESSHASH DECISION TREE

Input: vector x, split indices $j^1, \ldots, j^4$, split thresholds $v^1, \ldots, v^4$
$i \leftarrow 1$ // node index within level of tree
for $t \leftarrow 1$ **to** 4 **do**
  $v \leftarrow v_i^t$ // lookup split threshold for node i at level t
  $b \leftarrow x_{j^t} \geq v \; ? \; 1 : 0$ // above split threshold?
  $i \leftarrow 2i - 1 + b$ // assign to left or right child
end for
return i

ETH zürich

Compute Workload

- **Formula:**

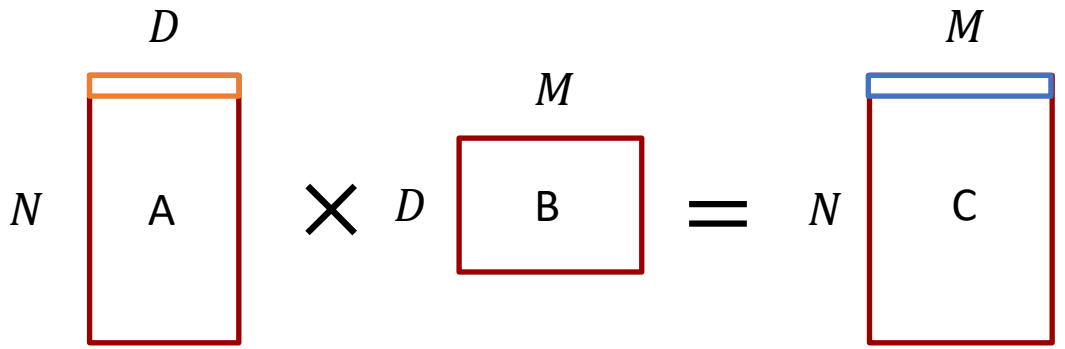
  $(AB)_{n,m} \approx \alpha\, dec\big(enc(A), h(B)\big)_{n,m} + \beta = \alpha \sum_{c=1}^{C} \boldsymbol{L_{c,k,m}} + \beta$ with $k = \boldsymbol{enc^{(c)}(a_n)}$

  with $A \in \mathbb{R}^{N \times D}$, $B \in \mathbb{R}^{D \times M}$, C the #codebooks, K the #learned prototypes

- **Computation & Lookups**
  - $enc(A)$: $\Theta(NC)$
  - $h(B)$: $\Theta(MKCD)$, typically learned **offline** → $\boldsymbol{L}$
  - $dec(\cdot,\cdot)$: $\Theta(NCM)$, with C table lookups for each output value (M cols, N rows)
  - overall: $\Theta\big(MC(KD+N)\big) \xrightarrow{K=16,\, N \gg D} \Theta(NCM)$

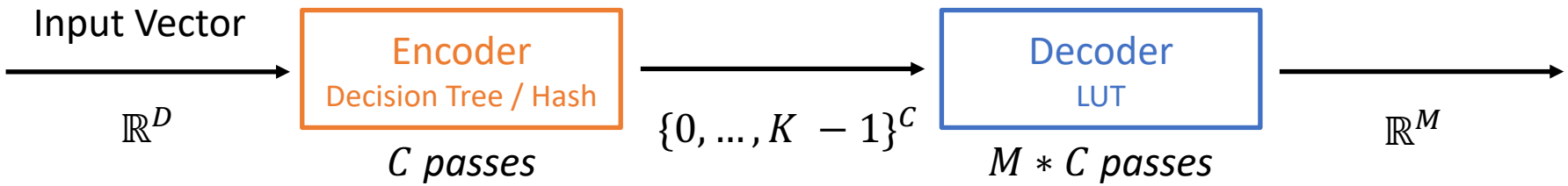
Overview

- **Overview Notation**
  - $A \in \mathbb{R}^{N \times D}$, $B \in \mathbb{R}^{D \times M}$
  - K = # prototypes, C = # codebooks
  - LUT → $L \in \mathbb{R}^{C \times K \times M}$



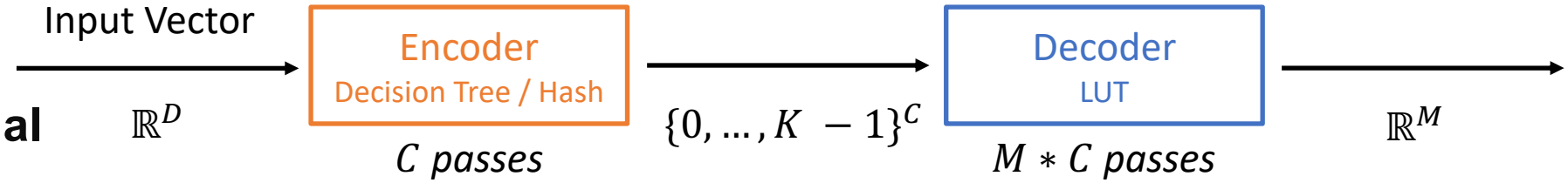
$$(AB)_{n,m} \approx \alpha \sum_{c=1}^{C} \boldsymbol{L_{c,k,m}} + \beta \text{ with } k = \boldsymbol{enc^{(c)}(a_n)}$$

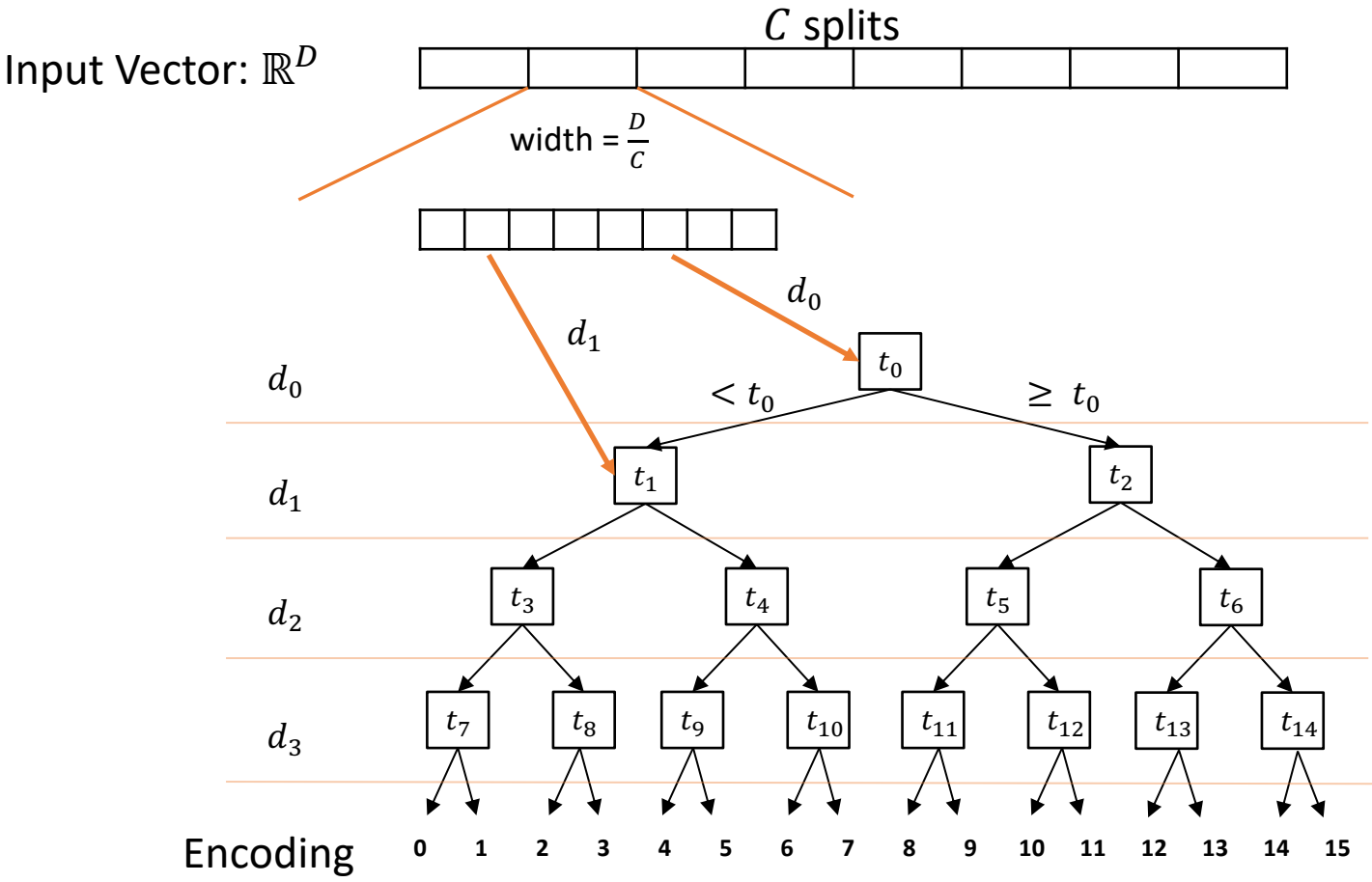
- **Overview Architecture**



Overview

- **Encoder Functional**

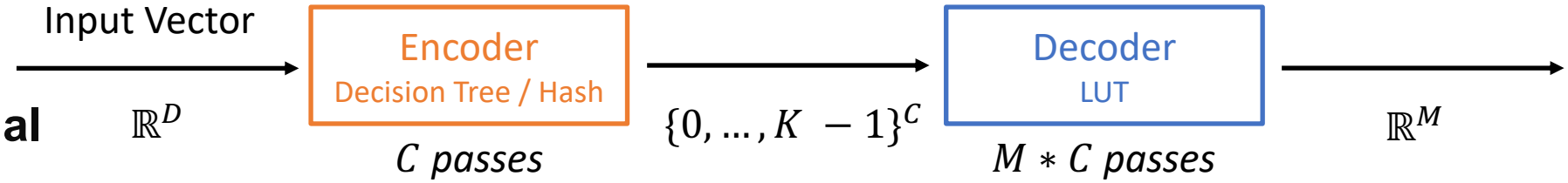


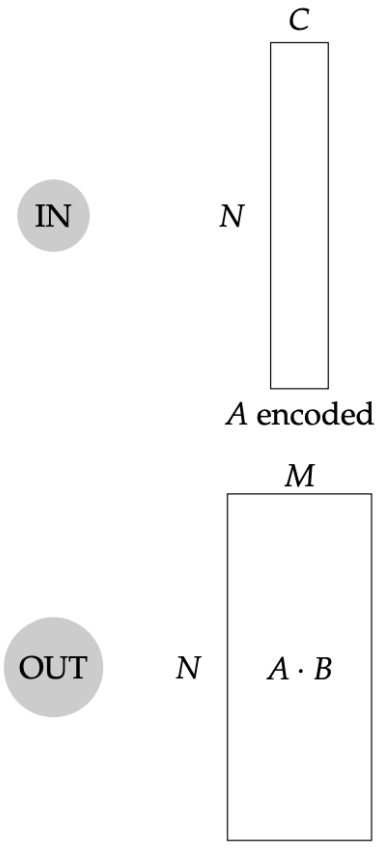


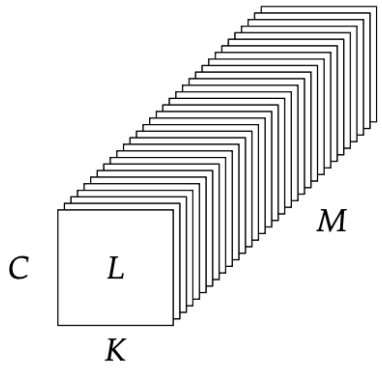
Name	Bits	Size
Thresholds (t_0 etc.)	dtype	$K-1$
Dimensions	$log_2(\frac{D}{C})$	$\sqrt[2]{K}$ (Madd), K (DT)
Remapping (only DT)	$log_2(K)$	K

Overview

- **Decoder Functional**







$A \in \mathbb{R}^{N \times D}$, $B \in \mathbb{R}^{D \times M}$
$L \in \mathbb{R}^{M \times C \times K}$ – LUTs
C – # of codebooks
K – # prototypes/codebook

$$(AB)_{n,m} \approx \alpha \sum_{c=1}^{C} L_{c,k,m} + \beta \text{ with } k = enc^{(c)}(a_n)$$

- Lookup and calculate sum over previously **encoded** index k

Algorithm 1: Decoding kernel

Data: $E \in \{0,\dots,K-1\}^K, L \in \mathbb{R}^{M \times C \times K}, M, C$
Result: $y \in \mathbb{R}^M$

```
m ← 0;
while m ≤ M do
    c ← 0;
    r ← 0;
    while c ≤ C do
        r ← r + L[m][c][E[c]];
        c ← c + 1;
    end
    y[m] ← r;
    m ← m + 1;
end
```

Results I

- **Table Explanation**

Name	[Out, In]	D	Accuracy drop [%]			FLOPs ↓ (RW ↓) [x]		LUT	Scaled error		
			Madd	DT	PQ	M&DT	PQ	[kB]	Madd	DT	PQ
1.0.conv2	[64, 64] (3x3)	576	-1.19	-1.64	-3.00	33.9 (1.0)	2.5 (0.1)	128	2.0E-02	2.4E-02	2.9E-02
1.1.conv3	[256, 64] (1x1)	64	-0.14	-0.28	-0.10	3.9 (1.0)	2.9 (0.2)	512	1.1E-02	1.5E-02	1.0E-02
2.0.conv1	[128, 256] (1x1)	256	-3.87	-3.66	-1.16	15.5 (1.0)	4.0 (0.1)	256	2.8E-02	2.7E-02	2.3E-02
2.0.conv2	[128, 128] (3x3)	1152	-8.45	-10.61	-2.65	69.8 (1.1)	5.0 (0.1)	256	4.7E-02	4.6E-02	3.7E-02

- All data is single layer replacement
- $C = 32$ and $K = 16$
- **FLOPs** in Maddness are only **compare and add**
- **RW** means off-chip read and write (so LUT lookups for Maddness are not counted)

Methods/ Encoding Functions:

- **M/Madd: Maddness** the in the paper described method: centroids are learned **top-down**
- **DT**: **Decision Tree** our method by building the centroid from the **bottom up** (nearly same hash function structure)
- **PQ**: Choosing the prototype closest the input vector **by MSE**

All function use the **same decoding** scheme

Results I

There would be more of these layers in ResNet-50

- **ResNet-50 ImageNet 1K (interesting layers)**

Name	[Out, In]	D	Accuracy drop [%]			FLOPs ↓ (RW ↓) [x]		LUT	Scaled error		
			Madd	DT	PQ	M&DT	PQ	[kB]	Madd	DT	PQ
1.0.conv2	[64, 64] (3x3)	576	-1.19	-1.64	-3.00	33.9 (1.0)	2.5 (0.1)	128	2.0E-02	2.4E-02	2.9E-02
1.1.conv3	[256, 64] (1x1)	64	-0.14	-0.28	-0.10	3.9 (1.0)	2.9 (0.2)	512	1.1E-02	1.5E-02	1.0E-02
2.0.conv1	[128, 256] (1x1)	256	-3.87	-3.66	-1.16	15.5 (1.0)	4.0 (0.1)	256	2.8E-02	2.7E-02	2.3E-02
2.0.conv2	[128, 128] (3x3)	1152	-8.45	-10.61	-2.65	69.8 (1.1)	5.0 (0.1)	256	4.7E-02	4.6E-02	3.7E-02
2.0.down.0	[512, 256] (1x1)	256	-6.91	-5.06	-3.88	15.9 (1.2)	9.1 (0.2)	1024	3.1E-02	2.7E-02	2.4E-02
2.3.conv3	[512, 128] (1x1)	128	-0.41	-0.51	-0.22	7.9 (1.1)	5.8 (0.3)	1024	1.3E-02	1.2E-02	1.0E-02
3.0.conv1	[256, 512] (1x1)	512	-9.79	-9.40	-2.51	31.5 (1.2)	8.0 (0.1)	512	2.6E-02	2.6E-02	2.2E-02
3.0.conv2	[256, 256] (3x3)	2304	-15.64	-16.27	-4.27	141.8 (2.2)	9.9 (0.1)	512	4.9E-02	4.9E-02	3.9E-02
3.0.conv3	[1024, 256] (1x1)	256	-3.65	-4.26	-2.23	15.9 (2.0)	11.6 (0.5)	2048	3.1E-02	3.2E-02	2.9E-02
3.3.conv3	[1024, 256] (1x1)	256	-0.71	-0.77	-0.38	15.9 (2.0)	11.6 (0.5)	2048	1.0E-02	1.0E-02	8.2E-03
3.4.conv2	[256, 256] (3x3)	2304	-2.65	-2.59	-1.52	141.8 (2.2)	9.9 (0.1)	512	1.9E-02	1.9E-02	1.6E-02
3.5.conv1	[256, 1024] (1x1)	1024	-2.42	-2.49	-1.55	63.0 (2.0)	9.1 (0.1)	512	3.9E-02	3.8E-02	3.2E-02
3.5.conv2	[256, 256] (3x3)	2304	-2.40	-2.42	-1.54	141.8 (2.2)	9.9 (0.1)	512	5.4E-02	5.6E-02	4.4E-02
4.0.conv1	[512, 1024] (1x1)	1024	-18.61	-17.97	-8.55	63.5 (2.7)	16.0 (0.2)	1024	5.1E-02	5.1E-02	4.3E-02
4.0.conv2	[512, 512] (3x3)	4608	-19.73	-20.78	-9.16	285.8 (10.4)	19.9 (0.7)	1024	6.9E-02	6.8E-02	5.7E-02
4.0.conv3	[2048, 512] (1x1)	512	-13.39	-14.71	-7.26	31.9 (9.4)	23.3 (2.2)	4096	4.0E-02	4.0E-02	3.6E-02
4.1.conv1	[512, 2048] (1x1)	2048	-12.50	-9.73	-4.95	127.0 (9.4)	18.3 (0.7)	1024	1.9E-02	1.9E-02	1.8E-02
4.2.conv1	[512, 2048] (1x1)	2048	-3.00	-2.79	-2.65	127.0 (9.4)	18.3 (0.7)	1024	8.7E-03	9.9E-03	8.4E-03
4.2.conv3	[2048, 512] (1x1)	512	-1.68	-1.64	-1.52	31.9 (9.4)	23.3 (2.2)	4096	4.5E-02	4.6E-02	3.8E-02

Results II

- **DS-CNN Google Speech v2**

3x-4x difference in scaled error

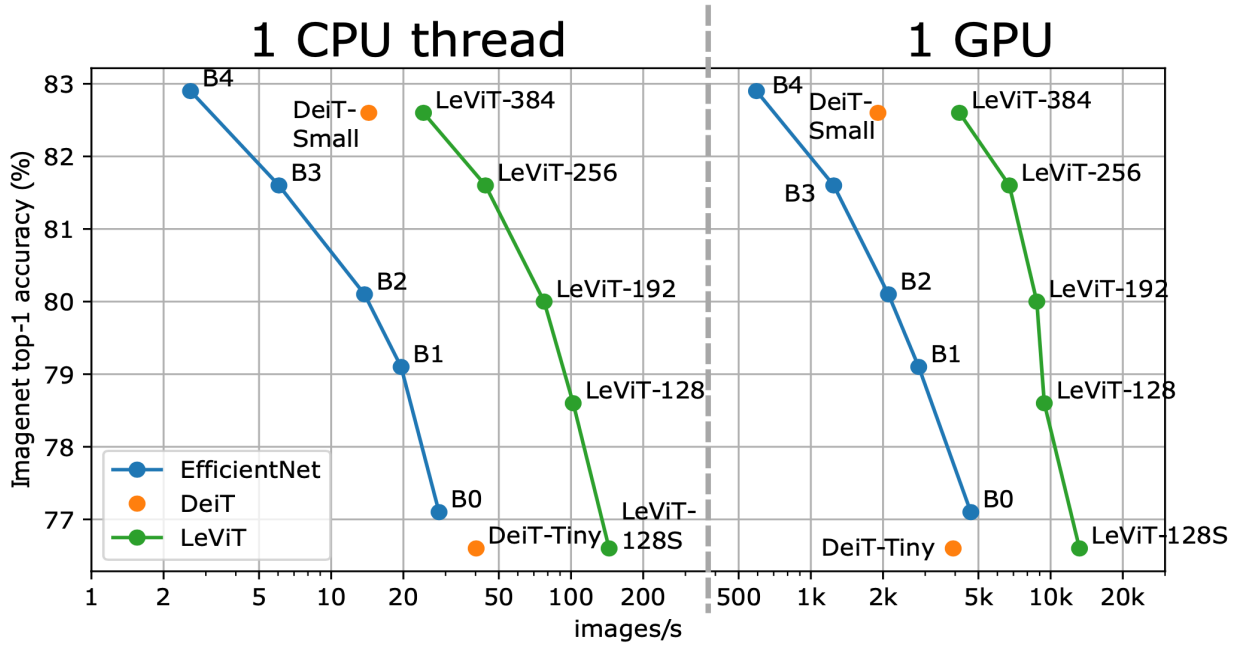
Name	[Out, In]	D	Accuracy drop [%]			FLOPs ↓ (RW ↓) [x]		LUT	Scaled error		
			Madd	DT	PQ	M&DT	PQ	[kB]	Madd	DT	PQ
conv1	[64, 1] (10x4)	40	-3.65	-8.89	-3.81	2.4 (1.0)	1.0 (0.1)	128	2.4E-03	3.6E-03	5.3E-03
conv2	[64, 1] (3x3)	9	-84.57	-84.57	-84.57	1.9 (1.0)	1.1 (0.3)	36	6.7E-02	6.7E-02	6.7E-02
conv3	[64, 64] (1x1)	64	-5.29	-8.28	-9.07	3.8 (1.0)	1.6 (0.1)	128	1.8E-02	2.0E-02	2.2E-02
conv4	[64, 1] (3x3)	9	-84.84	-84.77	-84.84	1.9 (1.0)	1.1 (0.3)	36	6.7E-02	6.7E-02	6.7E-02
conv5	[64, 64] (1x1)	64	-7.25	-10.62	-6.48	3.8 (1.0)	1.6 (0.1)	128	2.3E-02	2.9E-02	2.1E-02
conv6	[64, 1] (3x3)	9	-84.73	-84.77	-84.73	1.9 (1.0)	1.1 (0.3)	36	6.4E-02	6.4E-02	6.4E-02
conv7	[64, 64] (1x1)	64	-3.02	-8.55	-2.52	3.8 (1.0)	1.6 (0.1)	128	2.4E-02	3.3E-02	2.2E-02
conv8	[64, 1] (3x3)	9	-85.07	-85.11	-85.07	1.9 (1.0)	1.1 (0.3)	36	7.0E-02	7.0E-02	7.0E-02
conv9	[64, 64] (1x1)	64	-0.63	-2.50	-0.54	3.8 (1.0)	1.6 (0.1)	128	2.4E-02	3.5E-02	2.3E-02

- Convolution layers with groups=64 (conv2,conv4 etc) suffer from a very low D=9 (unrolled 3x3 kernel) resulting in a low “resolution”
- Possible results K^C (assuming all LUT values and sums are different)

Results III

- **LeViT SOTA transformer classifier**

Architecture	# params (M)	FLOPs (M)	inference speed top-1 %	GPU im/s	Intel im/s	ARM im/s	ImageNet -Real %	-V2. %
LeViT-128S **(ours)**	7.8	305	76.6	12880	131.1	39.1	83.1	64.3
EfficientNet B0	5.3	390	77.1	4754	30.1	3.5	83.5	64.3
LeViT-128 **(ours)**	9.2	406	78.6	9266	94.0	30.8	84.7	66.6
LeViT-192 **(ours)**	10.9	658	80.0	8601	65.0	24.2	85.7	68.0
EfficientNet B1	7.8	700	79.1	2882	20.0	2.3	84.9	66.9
EfficientNet B2	9.2	1000	80.1	2149	13.1	1.3	85.9	68.8
LeViT-256 **(ours)**	18.9	1120	81.6	6582	42.5	16.4	86.8	70.0
DeiT-Tiny	5.9	1220	76.6	3973	39.1	16.8	83.9	65.4
EfficientNet B3	12	1800	81.6	1272	5.9	0.8	86.8	70.6
LeViT-384 **(ours)**	39.1	2353	82.6	4165	23.1	9.4	87.6	71.3
EfficientNet B4	19	4200	82.9	606	2.5	0.5	88.0	72.3
DeiT-Small	22.5	4522	82.6	1931	13.7	7.6	87.8	71.7



[0]: LeViT: A Vision Transformer in ConvNet's Clothing for Faster Inference (Oct – 2021)

Instead of EfficientNet we opted to go with **LeViT-128S**.

Results IV

- **LeViT-128S**

Encoding/Hashing fails:
Both encoding algorithms are not able to learn a good encoding function

Scaled error:
Can be a useful indicator but not reliable

Pattern?:
No clear pattern emerges!

Name	[Out, In]	D	Accuracy drop [%]			FLOPs ↓ (RW ↓) [x]		LUT	Scaled error		
			Madd	DT	PQ	M&DT	PQ	[kB]	Madd	DT	PQ
0.m.proj.1	[128, 128]	128	-75.51	-76.10	-6.03	7.8 (1.3)	3.2 (0.1)	256	4.2E-02	4.4E-02	1.1E-02
0.m.qkv	[256, 128]	128	-75.76	-76.04	-3.21	7.9 (1.4)	4.6 (0.2)	512	2.3E-02	2.6E-02	7.4E-03
1.m.0	[256, 128]	128	-56.38	-51.97	-4.07	7.9 (1.4)	4.6 (0.2)	512	4.0E-02	4.2E-02	1.5E-02
1.m.2	[128, 256]	256	-55.19	-51.09	-1.56	15.5 (1.4)	4.0 (0.1)	256	4.6E-02	4.7E-02	1.8E-02
2.m.proj.1	[128, 128]	128	-45.64	-46.21	-1.16	7.8 (1.3)	3.2 (0.1)	256	8.5E-02	8.2E-02	2.5E-02
2.m.qkv	[256, 128]	128	-26.25	-23.99	-0.99	7.9 (1.4)	4.6 (0.2)	512	3.2E-02	3.2E-02	1.0E-02
3.m.0	[256, 128]	128	-40.76	-43.45	-4.35	7.9 (1.4)	4.6 (0.2)	512	3.8E-02	3.7E-02	1.5E-02
3.m.2	[128, 256]	256	-62.37	-61.79	-0.87	15.5 (1.4)	4.0 (0.1)	256	9.1E-02	7.8E-02	2.2E-02
4.kv	[640, 128]	128	-76.45	-76.39	-6.59	8.0 (1.5)	6.2 (0.4)	1280	4.0E-02	4.0E-02	1.5E-02
4.proj.1	[256, 512]	512	-76.49	-76.49	-19.23	31.5 (4.1)	8.0 (0.4)	512	7.9E-02	8.0E-02	4.3E-02
4.q.1	[128, 128]	128	-2.72	-2.65	-0.09	7.8 (2.2)	3.2 (0.2)	256	1.5E-01	1.5E-01	5.3E-02
5.m.0	[512, 256]	256	-4.01	-2.80	-0.77	15.9 (4.1)	9.1 (0.6)	1024	4.9E-02	5.0E-02	2.2E-02
5.m.2	[256, 512]	512	-33.21	-10.48	-0.50	31.5 (4.1)	8.0 (0.4)	512	1.5E-01	9.5E-02	4.3E-02
6.m.proj.1	[256, 192]	192	-9.29	-8.80	-0.71	11.8 (3.0)	5.6 (0.4)	512	9.2E-02	9.2E-02	3.5E-02
6.m.qkv	[384, 256]	256	-7.53	-6.99	-0.55	15.8 (3.8)	8.0 (0.5)	768	4.1E-02	4.1E-02	1.9E-02
7.m.0	[512, 256]	256	-4.29	-4.66	-1.11	15.9 (4.1)	9.1 (0.6)	1024	5.3E-02	5.1E-02	2.7E-02
7.m.2	[256, 512]	512	-10.72	-4.59	-0.63	31.5 (4.1)	8.0 (0.4)	512	1.2E-01	9.1E-02	4.5E-02
8.m.proj.1	[256, 192]	192	-4.24	-4.31	-0.43	11.8 (3.0)	5.6 (0.4)	512	4.1E-02	4.0E-02	1.7E-02
8.m.qkv	[384, 256]	256	-3.58	-3.36	-0.70	15.8 (3.8)	8.0 (0.5)	768	3.2E-02	3.2E-02	1.6E-02
9.m.0	[512, 256]	256	-6.08	-5.72	-1.50	15.9 (4.1)	9.1 (0.6)	1024	4.0E-02	3.9E-02	2.1E-02
9.m.2	[256, 512]	512	-9.05	-13.10	-0.65	31.5 (4.1)	8.0 (0.4)	512	6.4E-02	7.7E-02	3.3E-02
10.m.proj.1	[256, 192]	192	-7.04	-6.99	-0.70	11.8 (3.0)	5.6 (0.4)	512	6.3E-02	6.2E-02	2.4E-02
10.m.qkv	[384, 256]	256	-6.25	-5.33	-0.83	15.8 (3.8)	8.0 (0.5)	768	4.4E-02	4.5E-02	2.2E-02
11.m.0	[512, 256]	256	-9.75	-9.15	-2.32	15.9 (4.1)	9.1 (0.6)	1024	4.2E-02	4.1E-02	2.1E-02
11.m.2	[256, 512]	512	-49.63	-31.96	-1.18	31.5 (4.1)	8.0 (0.4)	512	9.8E-02	9.3E-02	3.9E-02
12.kv	[1280, 256]	256	-76.39	-76.32	-27.84	16.0 (4.9)	12.3 (1.3)	2560	5.7E-02	5.6E-02	3.2E-02
12.proj.1	[384, 1024]	1024	-76.47	-76.42	-54.33	63.3 (16.5)	12.8 (1.3)	768	9.7E-02	9.6E-02	6.0E-02
12.q.1	[256, 256]	256	-5.11	-4.50	-0.26	15.8 (8.1)	6.4 (0.9)	512	1.1E-01	1.0E-01	4.2E-02
13.m.0	[768, 384]	384	-7.00	-7.59	-3.24	23.9 (15.2)	13.7 (2.4)	1536	6.0E-02	5.9E-02	3.6E-02
13.m.2	[384, 768]	768	-18.77	-11.41	-1.72	47.5 (15.2)	12.0 (1.3)	768	2.0E-01	1.5E-01	8.2E-02
14.m.proj.1	[384, 256]	256	-3.02	-2.74	-0.95	15.8 (9.5)	8.0 (1.3)	768	8.6E-02	8.5E-02	4.9E-02
14.m.qkv	[512, 384]	384	-3.70	-3.77	-1.77	23.8 (13.2)	11.3 (1.7)	1024	6.9E-02	6.7E-02	4.3E-02
15.m.0	[768, 384]	384	-9.58	-8.33	-4.86	23.9 (15.2)	13.7 (2.4)	1536	5.6E-02	5.6E-02	3.8E-02
15.m.2	[384, 768]	768	-22.83	-14.72	-2.32	47.5 (15.2)	12.0 (1.3)	768	1.9E-01	1.5E-01	8.2E-02
16.m.proj.1	[384, 256]	256	-1.98	-2.39	-0.63	15.8 (9.5)	8.0 (1.3)	768	2.4E-02	2.6E-02	1.4E-02
16.m.qkv	[512, 384]	384	-2.92	-2.86	-2.34	23.8 (13.2)	11.3 (1.7)	1024	6.2E-02	6.2E-02	4.3E-02
17.m.0	[768, 384]	384	-8.15	-7.76	-5.45	23.9 (15.2)	13.7 (2.4)	1536	6.2E-02	6.1E-02	4.2E-02
17.m.2	[384, 768]	768	-20.68	-13.35	-3.27	47.5 (15.2)	12.0 (1.3)	768	1.3E-01	1.1E-01	6.7E-02
18.m.proj.1	[384, 256]	256	-3.41	-2.57	-0.92	15.8 (9.5)	8.0 (1.3)	768	2.0E-02	1.8E-02	9.7E-03
18.m.qkv	[512, 384]	384	-3.59	-4.42	-3.80	23.8 (13.2)	11.3 (1.7)	1024	5.8E-02	5.8E-02	4.1E-02
19.m.0	[768, 384]	384	-8.30	-13.50	-5.53	23.9 (15.2)	13.7 (2.4)	1536	8.2E-02	7.1E-02	4.3E-02
19.m.2	[384, 768]	768	-24.67	-10.81	-4.33	47.5 (15.2)	12.0 (1.3)	768	6.6E-02	5.6E-02	3.4E-02
20.m.proj.1	[384, 256]	256	-0.82	-1.09	-0.25	15.8 (9.5)	8.0 (1.3)	768	1.4E-02	1.3E-02	6.7E-03
20.m.qkv	[512, 384]	384	-0.60	-0.77	-0.62	23.8 (13.2)	11.3 (1.7)	1024	4.1E-02	4.1E-02	3.0E-02
21.m.0	[768, 384]	384	-3.98	-3.80	-3.38	23.9 (15.2)	13.7 (2.4)	1536	6.7E-02	6.1E-02	3.7E-02
21.m.2	[384, 768]	768	-16.37	-6.21	-2.89	47.5 (15.2)	12.0 (1.3)	768	1.5E-02	1.2E-02	7.8E-03

Takeaways

3x3 vs 1x1:
3x3 Conv2d are way harder than 1x1

Pattern:
- Loosely correlated to reduction in FLOPs
- Loosely correlated to scaled error
- But often **hard to understand and predict** how well Maddness works

What is possible on the algorithm side?
- We could already define the upper limit in accuracy by using the PQ method ("perfect encoding") and run the experiments with $C = D$

Is it worth to build hardware for it?
- Probably not, especially for SOTA models that are already **highly optimized** in terms of parameters

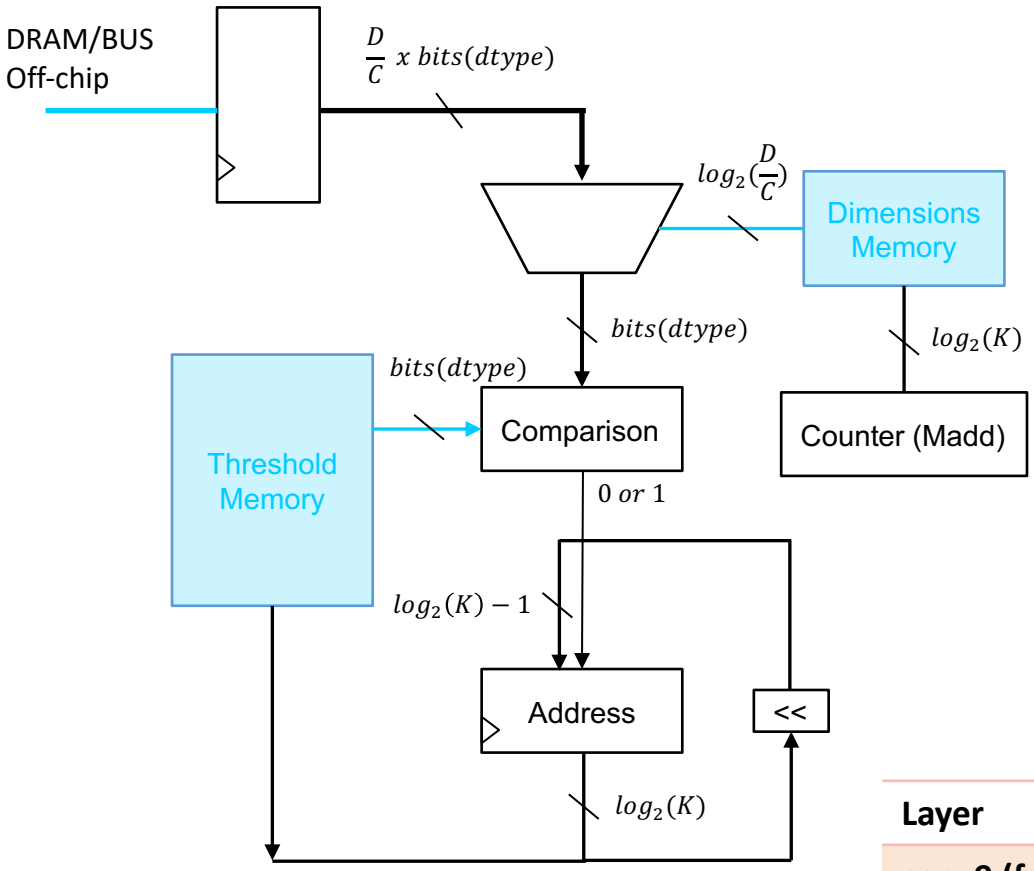
Name	[Out, In]	D	Accuracy drop [%]			FLOPs ↓ (RW ↓) [x]		LUT	Scaled error		
			Madd	DT	PQ	M&DT	PQ	[kB]	Madd	DT	PQ
0.m.proj.1	[128, 128]	128	-75.51	-76.10	-6.03	7.8 (1.3)	3.2 (0.1)	256	4.2E-02	4.4E-02	1.1E-02
0.m.qkv	[256, 128]	128	-75.76	-76.04	-3.21	7.9 (1.4)	4.6 (0.2)	512	2.3E-02	2.6E-02	7.4E-03
1.m.0	[256, 128]	128	-56.38	-51.97	-4.07	7.9 (1.4)	4.6 (0.2)	512	4.0E-02	4.2E-02	1.5E-02
1.m.2	[128, 256]	256	-55.19	-51.09	-1.56	15.5 (1.4)	4.0 (0.1)	256	4.6E-02	4.7E-02	1.8E-02
2.m.proj.1	[128, 128]	128	-45.64	-46.21	-1.16	7.8 (1.3)	3.2 (0.1)	256	8.5E-02	8.2E-02	2.5E-02
2.m.qkv	[256, 128]	128	-26.25	-23.99	-0.99	7.9 (1.4)	4.6 (0.2)	512	3.2E-02	3.2E-02	1.0E-02
3.m.0	[256, 128]	128	-40.76	-43.45	-4.35	7.9 (1.4)	4.6 (0.2)	512	3.8E-02	3.7E-02	1.5E-02
3.m.2	[128, 256]	256	-62.37	-61.79	-0.87	15.5 (1.4)	4.0 (0.1)	256	9.1E-02	7.8E-02	2.2E-02
4.kv	[640, 128]	128	-76.45	-76.39	-6.59	8.0 (1.5)	6.2 (0.4)	1280	4.0E-02	4.0E-02	1.5E-02
4.proj.1	[256, 512]	512	-76.49	-76.49	-19.23	31.5 (4.1)	8.0 (0.4)	512	7.9E-02	8.0E-02	4.3E-02
4.q.1	[128, 128]	128	-2.72	-2.65	-0.09	7.8 (2.2)	3.2 (0.2)	256	1.5E-01	1.5E-01	5.3E-02
5.m.0	[512, 256]	256	-4.01	-2.80	-0.77	15.9 (4.1)	9.1 (0.6)	1024	4.9E-02	5.0E-02	2.2E-02
5.m.2	[256, 512]	512	-33.21	-10.48	-0.50	31.5 (4.1)	8.0 (0.4)	512	1.5E-01	9.5E-02	4.3E-02
6.m.proj.1	[256, 192]	192	-9.29	-8.80	-0.71	11.8 (3.0)	5.6 (0.4)	512	9.2E-02	9.2E-02	3.5E-02
6.m.qkv	[384, 256]	256	-7.53	-6.99	-0.55	15.8 (3.8)	8.0 (0.5)	768	4.1E-02	4.1E-02	1.9E-02
7.m.0	[512, 256]	256	-4.29	-4.66	-1.11	15.9 (4.1)	9.1 (0.6)	1024	5.3E-02	5.1E-02	2.7E-02
7.m.2	[256, 512]	512	-10.72	-4.59	-0.63	31.5 (4.1)	8.0 (0.4)	512	1.2E-01	9.1E-02	4.5E-02
8.m.proj.1	[256, 192]	192	-4.24	-4.31	-0.43	11.8 (3.0)	5.6 (0.4)	512	4.1E-02	4.0E-02	1.7E-02

Recommended next steps on the algorithmic side:
- $C = D$ runs to get the **upper limit for the accuracy** disregarding the resulting LUT size (TODO: fix CUDA kernel for high C)
- Look into **retraining** starting with DS-CNN and see how much there is to gain

Encoder Unit

Example Layers:

Name	[Out, In]	D	Accuracy drop [%] Madd	DT	PQ	FLOPs ↓ (RW ↓) [x] M&DT	PQ	LUT [kB]	Scaled error Madd	DT	PQ
conv9	[64, 64] (1x1)	64	-0.63	-2.50	-0.54	3.8 (1.0)	1.6 (0.1)	128	2.4E-02	3.5E-02	2.3E-02
15.m.2	[384, 768]	768	-22.83	-14.72	-2.32	47.5 (15.2)	12.0 (1.3)	768	1.9E-01	1.5E-01	8.2E-02
20.m.qkv	[512, 384]	384	-0.60	-0.77	-0.62	23.8 (13.2)	11.3 (1.7)	1024	4.1E-02	4.1E-02	3.0E-02



Memory sizes:

Name	Bits	Amount	Conv9 (fp16)	20.m.qkv (fp16)	15.m.2 (fp16)
Thresholds	dtype	$K-1$	30 Bytes	30 Bytes	30 Bytes
Dimensions	$log_2(\frac{D}{C})$	$\sqrt[2]{K}$ (Madd)	4 Bits	16 Bits	96 Bits

- Address after 4 cycles = **encoded prototype**
- On-chip memory size and bandwidth shouldn't be a problem
- Off-chip **memory bandwidth** as with normal GEMM
- Up to C=32 units in **parallel,** one per codebook

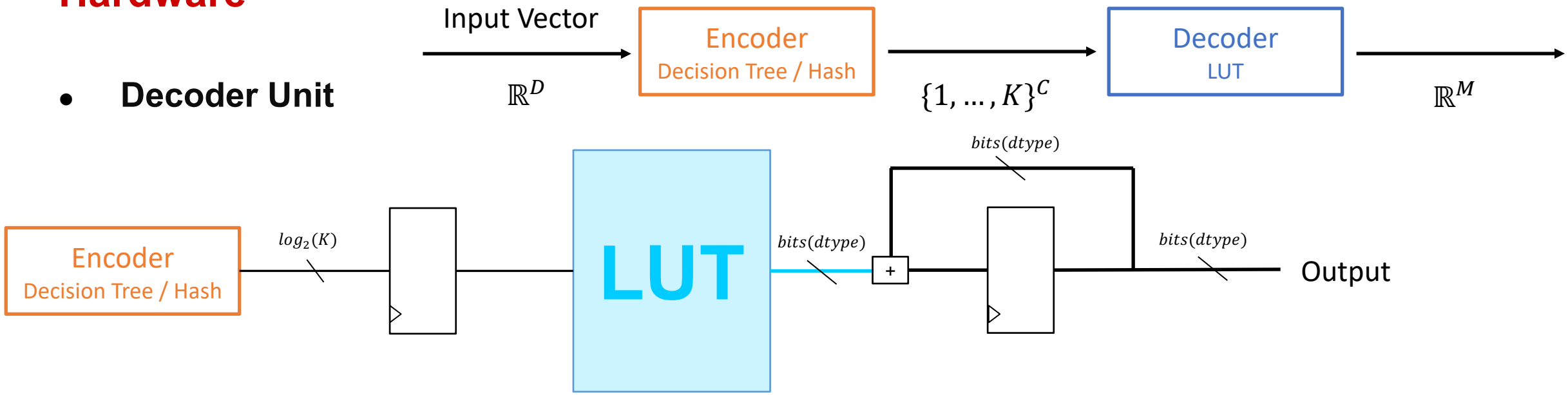
Memory bandwidth needed per unit:

- $\frac{D}{C}$ x $bits(dtype)$ per encoding

Layer	BW single unit/encoding	BW 32 units/encoding
conv9 (fp16)	4 Bytes	128 Bytes
20.m.qkv (fp16)	24 Bytes	768 Bytes
15.m.2 (fp16)	48 Bytes	1536 Bytes

Hardware

- **Decoder Unit**



- Time multiplex over **M to reduce LUT size,** tradeoffs + strategies discussed later
- NDA not signed → area estimated with 1 GE per SRAM bit (gf22: 0.199 µm2, tsmc65: 1.28µm2)

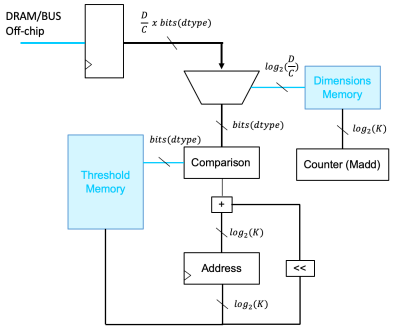
Layer	LUT [C, K] "smallest"	LUT [C, K, M] "all"	Area [C, K] Gf22 (tsmc65)	Area [C, K, M] Gf22 (tsmc65)
conv9 (fp16)	1 kB	64 kB	$0.0016mm^2$ ($0.010mm^2$)	$0.104mm^2$ ($0.671mm^2$)
20.m.qkv (fp16)	1 kB	512 kB	$0.0016mm^2$ ($0.010mm^2$)	$0.835mm^2$ ($5.369mm^2$)
15.m.2 (fp16)	1 kB	384 kB	$0.0016mm^2$ ($0.010mm^2$)	$0.626mm^2$ ($4.027mm^2$)

Example Layers:

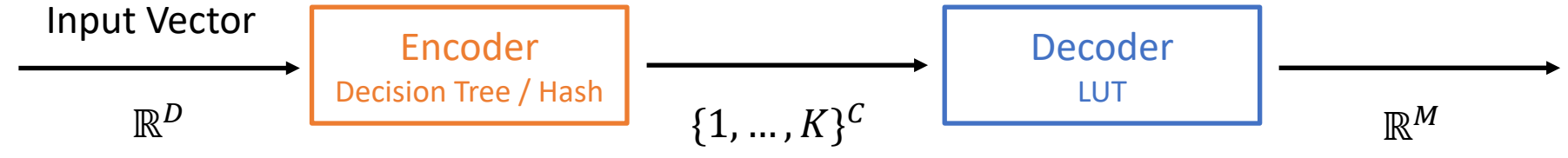
Name	[Out, In]	D	Accuracy drop [%] Madd	DT	PQ	FLOPs ↓ (RW ↓) [x] M&DT	PQ	LUT [kB]	Scaled error Madd	DT	PQ
conv9	[64, 64] (1x1)	64	-0.63	-2.50	-0.54	3.8 (1.0)	1.6 (0.1)	128	2.4E-02	3.5E-02	2.3E-02
15.m.2	[384, 768]	768	-22.83	-14.72	-2.32	47.5 (15.2)	12.0 (1.3)	768	1.9E-01	1.5E-01	8.2E-02
20.m.qkv	[512, 384]	384	-0.60	-0.77	-0.62	23.8 (13.2)	11.3 (1.7)	1024	4.1E-02	4.1E-02	3.0E-02

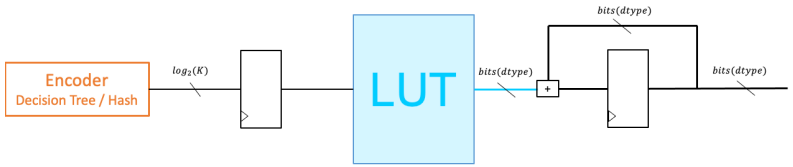
Hardware

- **Scale-out**



=

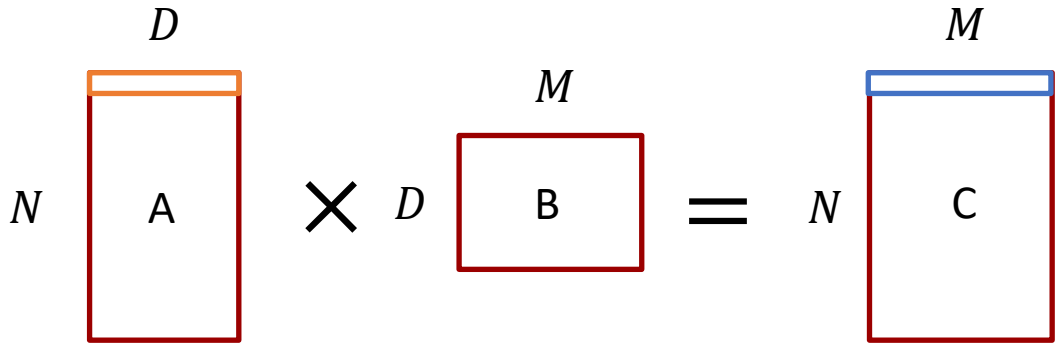




=

Hardware - Parallelize

- **Overview Notation**
  - $A \in \mathbb{R}^{N \times D}$, $B \in \mathbb{R}^{D \times M}$
  - K = # prototypes, C = # codebooks
  - LUT → $L \in \mathbb{R}^{C \times K \times M}$

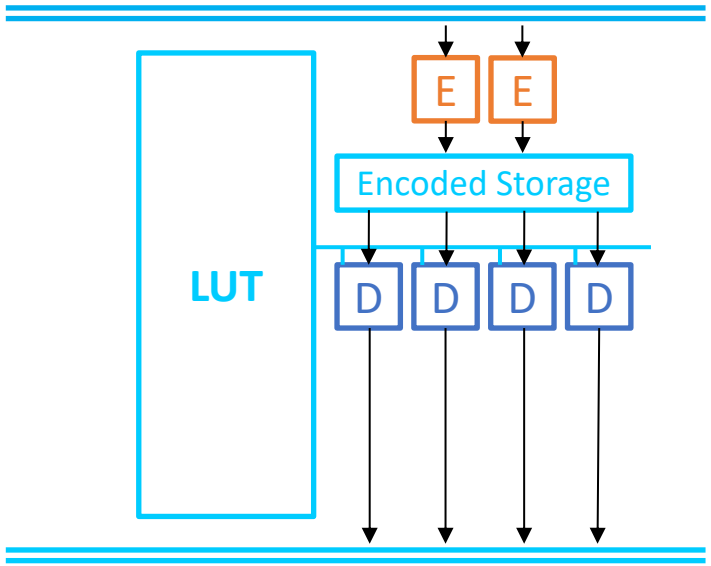


$$(AB)_{n,m} \approx \alpha \sum_{c=1}^{C} L_{c,k,m} + \beta \text{ with } k = enc^{(c)}(a_n)$$

- **Dimensions to parallelize (our examples C=32, K=16)**
- We have four possible dimensions $[N, C\ e.g\ D, K, M]$
  - **K** not possible as we would split up encoding unit and summing of decoder
  - **M** can be used to reduce LUT size by time multiplexing (requires multiple reloads of LUTs)
    - But also parallelize with the decoder (reuse of encoding)
  - **C** trivial for encoding, intermediate **encoding storage** needed for decoding
  - **N** would need duplication of LUT!

Hardware

- **Scale-out III / Parallelize over C (encoding) and M (decoding)**



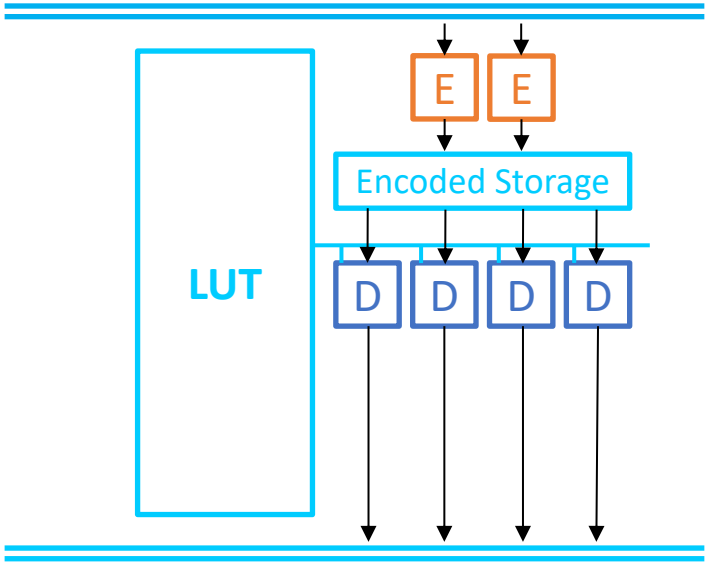
- Total area estimate still approximately the same as LUT will take up most of the space
- Memory hierarchy in general still needs to be discussed

- **Encoded Storage + LUT** size determine reuse e.g. reloads needed
- Proposed strategies reload LUT and never the input matrix A (**no double encoding**)

Layer	N	Encoded storage = N * C * $log_2(K)$
conv9 (fp16)	125	16kGE
20.m.qkv (fp16)	18	2.3kGE
15.m.2 (fp16)	18	2.3kGE

Hardware

- **Scale-out III / Parallelize over C (encoding) and M (decoding)**
- **Encoded Storage + LUT** size determine reuse e.g. reloads needed
- Proposed strategies reload LUT and never the input matrix A (**no double encoding**)



- Standalone or integrate into PULP?
- **ISA extension** probably easiest
- What about pipelining, vectorization?

M	LUT	LUT reloads conv9 (fp16)	LUT reloads 20.m.qkv (fp16)	LUT reloads 15.m.2 (fp16)	Area LUT Gf22 (tsmc65)
1	1 kB	63	511	383	$0.0016mm^2$ ($0.010mm^2$)
16	16 kB	3	31	23	$0.026mm^2$ ($0.168mm^2$)
64	64 kB	NONE	7	5	$0.104mm^2$ ($0.671mm^2$)
128	128 kB	NONE	3	2	$0.208mm^2$ ($1.342mm^2$)
512	512 kB	NONE	NONE	NONE	$0.835mm^2$ ($5.369mm^2$)

Example Layers:

Name	[Out, In]	D	Accuracy drop [%]			FLOPs ↓ (RW ↓) [x]		LUT	Scaled error		
			Madd	DT	PQ	M&DT	PQ	[kB]	Madd	DT	PQ
conv9	[64, 64] (1x1)	64	-0.63	-2.50	-0.54	3.8 (1.0)	1.6 (0.1)	128	2.4E-02	3.5E-02	2.3E-02
15.m.2	[384, 768]	768	-22.83	-14.72	-2.32	47.5 (15.2)	12.0 (1.3)	768	1.9E-01	1.5E-01	8.2E-02
20.m.qkv	[512, 384]	384	-0.60	-0.77	-0.62	23.8 (13.2)	11.3 (1.7)	1024	4.1E-02	4.1E-02	3.0E-02

Conclusion

- Not much left to gain on algorithmic side. **It is as good as it is.**
  - C=D run to get upper limit
  - retraining
- Hardware
  - Standalone or ISA extension?