Supplementary Material

1. Detailed Running Speed

We give detailed report on decoding time on our PyTorch implementation. Extending Table 2 and Table 3 in the main body, latency for each process averaged on Kodak and Tecnick is shown in Table 1. It is apparent that for all serial models, calculating context features and entropy parameters is the bottleneck, which takes more than 95% decoding time. By adopting the proposed parallel context model, Minnen2018's average decoding speed increases 44.6 times on Kodak (small images) and 52.1 times on Tecnick (large images). Meanwhile, Cheng2020 also speeds up 18.5 and 20.4 times tested on the two datasets due to a parallel context model.

We also test above mentioned models on CPUs as a reference (see Table 2). Though we mainly aim to improve the decoding efficiency on parallel devices, our parallel context model can still perform well on CPUs because of a sufficient use of matrix librarys *e.g.* Intel MKL, which is very encouraging. We measure the speeds by running our PyTorch implementation on Intel Xeon E5-2620.

2. More RD Curves

For completeness, we evaluate all models on Tecnick. Figure 4 shows the results. Compared with a channel conditioned solution Minnen2020, Cheng2020 with proposed checkerboard context model performs slightly worse in the low bit rate but better in the high bit rate. This perhaps because GMM cannot estimate the density precisely when using less coding channels (N = 128 in our case), as described in the original Cheng2020 paper. On the other hand, Minnen2018 with checkerboard context model even slightly outperforms original Minnen2018 baseline in higher bit rates. It further proves that a checkerboard-shaped convolution can extract more causal relationship and helps save more rate, especially in larger images which have more smooth area with more spatial redundancy.

We also optimize our models for MS-SSIM with $D = 1 - \mathrm{MSSSIM}(\boldsymbol{x}, \hat{\boldsymbol{x}})$ and test all models with MS-SSIM. To train models for MS-SSIM optimization, we set λ to $\{1, 3, 16\}$ for low bit rate models (N = 128) and $\{40, 120, 360\}$ for high rate models (N = 192). Figure 5 shows the results tested on Kodak.

3. Cheap Operations for Parallel En/De-coding

3.1. Multiplexer: Generating $\hat{\boldsymbol{y}}_{\text{half}}$ from $\hat{\boldsymbol{y}}$

As described in the main body, we generate $\hat{\boldsymbol{y}}_{\text{half}}$ and $\boldsymbol{b}_{\text{half}}$ to perform a one-pass encoding. To implement it we further define a multiplexer operation MUX:

$$\mathrm{MUX}(\boldsymbol{\alpha}, \boldsymbol{\beta})_i = \begin{cases} \alpha_i, & \hat{y}_i \in \hat{\boldsymbol{y}}_{\text{anchor}} \\ \beta_i, & otherwise \end{cases}$$

where inputs $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are feature maps with the same sizes $H \times W \times M$ as $\hat{\boldsymbol{y}}$. So we can simply calculate $\hat{\boldsymbol{y}}_{\text{half}}$ and $\boldsymbol{b}_{\text{half}}$ with below formula:

$$\hat{\boldsymbol{y}}_{\text{half}} = \mathrm{MUX}(\hat{\boldsymbol{y}}, \boldsymbol{0})$$

$$\boldsymbol{b}_{\text{half}} = \mathrm{MUX}(\boldsymbol{b}_{\text{map}}, \boldsymbol{0})$$

where $\boldsymbol{b}_{\text{map}} \in \mathbb{R}^{H \times W \times M}$ is the feature map with every $H \times W$ locations filled with the M-element bias vector $\boldsymbol{b}$.

Since we do not want the newly introduced multiplexer operation slow down the encoding process, it must be elaborately arranged as a series of light operations like slicing or viewing operators in PyTorch. We provide two of practical ways to implement the multiplexer.

3.1.1 Using slice-assign operations

For frameworks such as PyTorch and TensorFlow supporting slice-and-assign operators, we simply use such operators with a slicing step of 2 to mix the anchors and non-anchors input into the multiplexer. It could be a piece of code like:

```
# mix = MUX(alpha, beta)
mix = clone(alpha)
mix[..., 1::2, 0::2] = beta[..., 1::2, 0::2]
mix[..., 0::2, 1::2] = beta[..., 0::2, 1::2]
```

3.1.2 Using slicing and concatenating

For frameworks which do not support slice-assign operations, also we could mix anchors and non-anchors with shape-transforming operations like reshaping, permuting, slicing or concatenating. See Figure 1. Firstly, we flatten each 2×2 patches using a space-to-depth operator (or

architecture (N=192)		hyper synthesis	parameter calculation	%	latent synthesis	total (ms)	speed (Mpps)
Kodak, image size: 768×512, feature size: $48 \times 32 \times M$							
Ballé2018		1.30	-	-	25.04	26.34	14.93
Minnen2018	w/o context	1.32	2.23	8.4%	22.85	26.41	14.89
	serial	1.26	1302.42	98.4%	20.98	1323.66	0.30
	parallel (ours)	1.42	4.75	16.0%	23.49	29.66	13.26
Cheng2020	serial	1.82	1325.32	95.0%	68.21	1395.35	0.28
	parallel (ours)	1.72	4.37	5.8%	69.14	75.23	5.23
Tecnick, image size: 1200×1200, feature size: $75 \times 75 \times M$							
Ballé2018		2.22	-	-	81.06	83.28	17.29
Minnen2018	w/o context	4.01	1.30	1.5%	81.00	86.31	16.68
	serial	3.78	4891.43	98.3%	82.77	4977.98	0.29
	parallel (ours)	4.27	8.36	8.8%	82.83	95.46	15.08
Cheng2020	serial	3.40	5044.93	95.3%	247.83	5296.16	0.27
	parallel (ours)	3.09	10.98	4.2%	245.30	259.37	5.55

Table 1. Running time of each decoding process (unit: microsecond) on GPU. Meaning of table headers is the same as which is in Table 2 and Table 3 in the main body. The column *speed* shows million-pixels-per-second (Mpps) of each model, which represents how fast a specific architecture can decode images from bitstream.

architecture (N=192)		hyper synthesis	parameter calculation	%	latent synthesis	total (s)	speed (Kpps)
Kodak, image size: 768×512, feature size: $48 \times 32 \times M$							
Ballé2018		0.059	-	-	1.460	1.519	258.89
Minnen2018	w/o context	0.118	0.009	0.5%	1.720	1.848	212.79
	serial	0.115	20.402	99.4%	1.875	20.518	19.23
	parallel (ours)	0.113	0.083	3.9%	1.937	2.132	184.42
Cheng2020	serial	0.108	22.107	91.7%	1.901	24.116	16.30
	parallel (ours)	0.095	0.086	4.6%	1.678	1.859	211.49

Table 2. Running time of each decoding process (unit: second) on CPU.

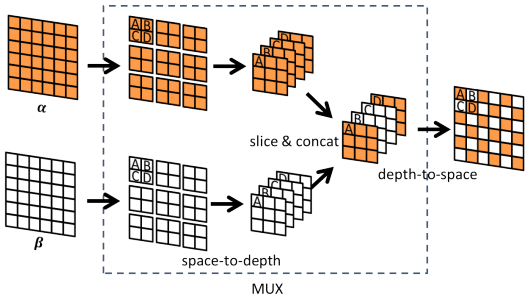

Figure 1. Multiplexer implementation with slicing and concatenating operators.

a series of permuting and reshaping) and then slice every feature maps, anchor and non-anchor, into four chunks and concatenate four of eight chunks to get what we require. Since most popular frameworks support above mentioned operators, this provides a more general solution to implement the proposed checkerboard model.

3.2. Demultiplexer: Split Latents for Encoding

During encoding, latents must get divided into two chunks: anchors and non-anchors, and then be encoded by AE separately. Otherwise, future decoding will fail because AD must read all anchors from the bitstream before decoding non-anchors. So we also require a cheap operation to separate latents into anchors and non-anchors. We use an inverse operation of slice-and-concatenate multiplexer introduced in Section 3.1.2 and Figure 1. See Figure 2, this new operation called demultiplexer performs a space-to-depth and then slices the latents or entropy parameters along channels to get anchors and non-anchors (or their corresponding entropy parameters).

3.3. Merging Two Chunks During Decoding

Figure 2 also shows how to decode and recover the two chunks of latents from the bitstream (right part in the diagram). With it we further discuss how we implement the two-pass decoding. After viewing, filler chunks will get inserted into the decoded feature maps via slicing and con-

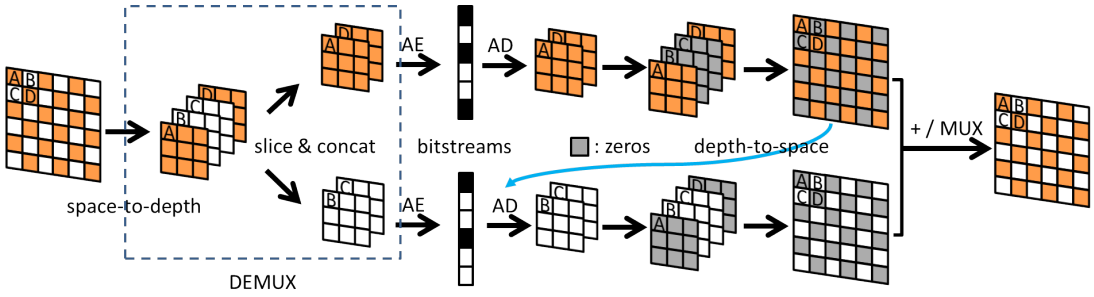

Figure 2. Demultiplexer (DEMUX) implementation with slicing and concatenating operators. Anchors and non-anchors are decoded from bitstream in turn and finally get gathered.

catenating (in our implementation, we simply use zero tensors as fillers). Then we reshape the filled feature maps to get $\hat{\boldsymbol{y}}_{\text{anchor}}$ or $\hat{\boldsymbol{y}}_{\text{non}-\text{anchor}}$ with sizes of $H \times W \times M$, as is described in Figure 5 in the main body. After finally decoding all anchors and non-anchors, we merge the two chunks with above mentioned multiplexer (or an addition operation if using zero fillers).

(a) ground truth (b) serial (BPP = 0.219, PSNR = 31.96) (c) parallel (BPP = 0.227, PSNR = 31.80)

(d) ground truth (e) serial (BPP = 0.449, PSNR = 33.54) (f) parallel (BPP = 0.454, PSNR = 33.42)

(g) ground truth (h) serial (BPP = 0.091, PSNR = 28.29) (i) parallel (BPP = 0.095, PSNR = 28.01)

Figure 3. Reconstructed images *kodim15*, *kodim02* and *kodim09* from Kodak. The *serial* ones are evaluated on our reproduced Cheng2020 models optimized for MSE and the *parallel* ones are from Cheng2020 with proposed checkerboard context model optimized for MSE.

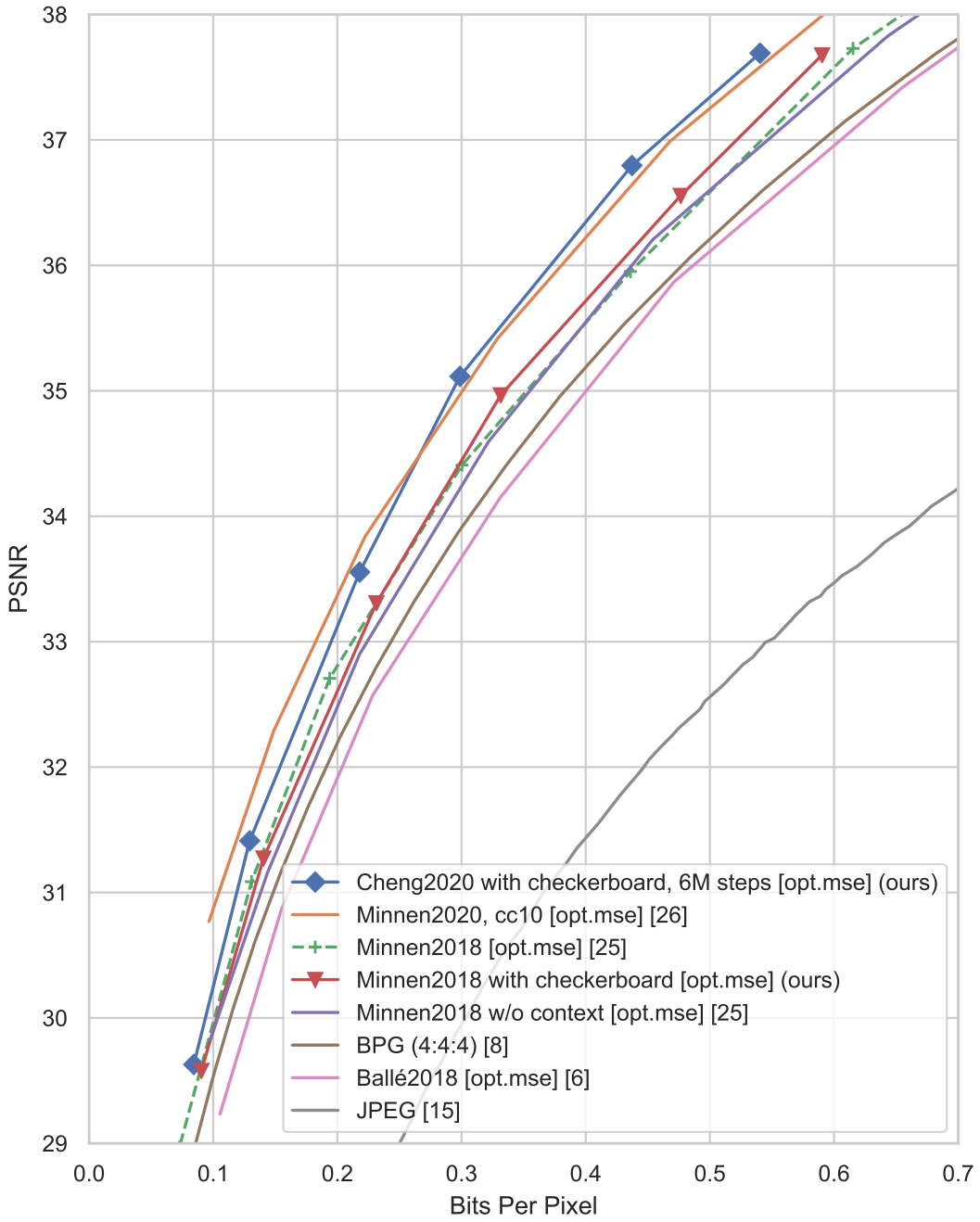

Figure 4. PSNR evaluation on Tecnick. Dash lines represent models adopting the serial context model.

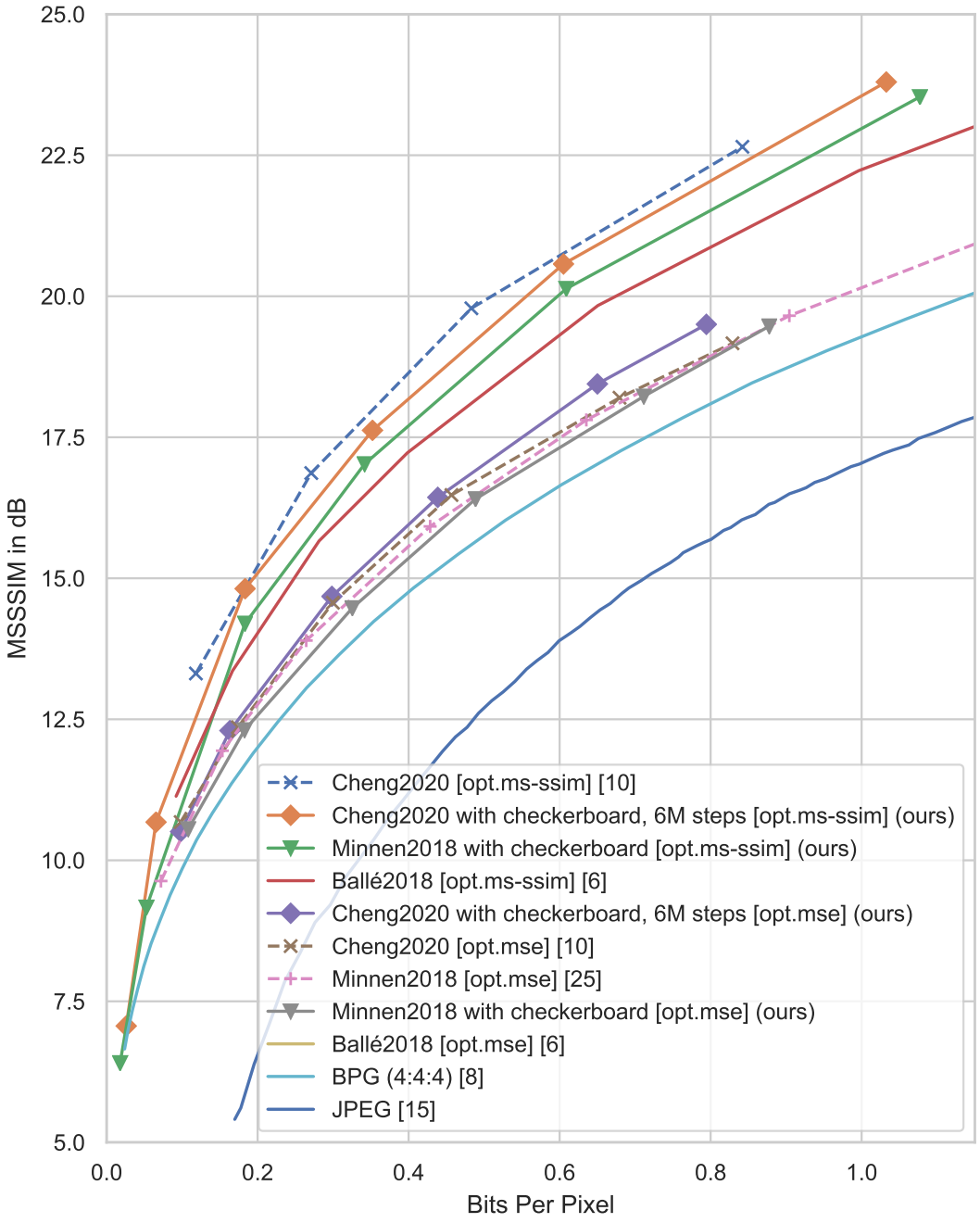

Figure 5. MS-SSIM evaluation on Kodak. Results are converted to decibels: $-10\log_{10}(1-\mathrm{MSSSIM}(x,\hat{x}))$. Dash lines represent models adopting the serial context model.