A Supplementary Material

A.1 Training Details

To ensure stable training, we applied gradient clipping with a maximum norm of 1.0 and used the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$ Kingma & Ba (2015). We used the built-in polynomial decay learning rate scheduler in MetaSeq with 500 warmup updates and the end learning rate set to 0. All models are trained with pre-norm and using ReLU activation. We apply a dropout of 0.1 throughout, but we do not apply any dropout to embeddings. We also use weight decay of 0.1. To initialize the weights, we use a variant based on Megatron-LM codebase, which involves using a normal distribution with a mean of zero and a standard deviation of 0.006. We truncate this normal distribution within two standard deviations and observed substantial gain in both training stability and performance.

A.2 Motivation

Why is the local model needed? Many of the efficiency advantages of the MEGABYTE design could be realized with the Global model alone, which would resemble a decoder version of ViT (Dosovitskiy et al., 2020). However, the joint distribution over the patch $p(x_{t+1}, .., x_{t+P}|x_{0..t})$ has an output space of size 256^P so direct modeling is only tractable for very small patches. We could instead factor the joint distribution into conditionally independent distributions $p(x_{t+1}|x_{0..t})..p(x_{t+P}|x_{0..t})$, but this would greatly limit the model's expressive power. For example, it would be unable to express a patch distribution such as 50% *cat* and 50% *dog*, and would instead have to assign probability mass to strings such as *cag* and *dot*. Instead, our autoregressive Local model conditions on previous characters within the patch, allowing it to only assign probability to the desired strings.

Increasing Parameters for Fixed Compute Transformer models have shown consistent improvements with parameter counts (Kaplan et al., 2020). However, the size of models is limited by their increasing computational cost. MEGABYTE allows larger models for the same cost, both by making self attention sub-quadratic, and by using large feedforward layers across patches rather than individual tokens.

Re-use of Established Components MEGABYTE consists of two transformer models interleaved with shifting, reshaping and a linear projection. This re-use increases the likelihood that the architecture will inherit the desirable scaling properties of transformers.

A.3 Model Details

As discussed in Section 4, we conduct experiments using a fixed compute and data budget across all models to focus our comparisons solely on the model architecture rather than training resources. To achieve this, we adjust model hyperparameters within each architecture so that the time taken for a single update is matched and then train all models for the same number of updates. We list all of model details in Table 8 and Table 9.

	Model	#L	d_{model}	#H	d_{head}
S1	125M	12	768	12	64
S2	350M	24	1024	16	64
S3	760M	24	1536	16	96
S4	1.3B	24	2048	32	64
S5	2.7B	32	2560	32	80
S6	6.7B	32	4096	32	128

Table 8: **Common Model architecture details by size.** For each model size, we show the number of layers (#L), the embedding size (d_{model}), the number of attention heads (#H), the dimension of each attention head (d_{head}).

Model	(Global) Size	Local Size	BS	LR	Context Length (in bytes)
arXiv					
Transformer	320M (D=1024, L=22)	N/A	72	2.00E-04	1,024
Perceiver AR	248M (D=1024, L=17)	N/A	72	2.00E-04	8,192 (1024 latents)
MEGABYTE	758M (D=2048, L=14)	262M (D=1024, L=18)	48	2.00E-04	8,192 (patch size 8)
w/o Local model	2.3B (D=2560, L=20)	N/A	48	1.50E-04	8,192 (patch size 4)
w/o global model	N/A	350M (D=1024, L=24)	192	2.00E-04	8,192 (patch size 8)
w/o cross-patch Local model	921M (D=2048, L=17)	350M (D=1024, L=24)	48	2.00E-04	8,192 (patch size 8)
w/ CNN encoder	704M (D=2048, L=13)	262M (D=1024, L=18)	48	2.00E-04	8,192 (patch size 8)
Image task 64 (Table 2)					
MEGABYTE	2.7B (D=2560, L=32)	350M (D=1024, L=24)	2	2.00E-04	12,288 (patch size 12)
Image task 64 (Table 4)					
Transformer	760M (D=1536, L=24)	N/A	512	3.00E-04	2,048
Perceiver AR	227M (D=1024, L=16)	N/A	512	3.00E-04	12,288 (1024 latents)
MEGABYTE	1.3B (D=2048, L=24)	1.3B (D=2048, L=24)	256	3.00E-04	12,288 (patch size 12)
Image task 256					
Transformer	62M (D=768, L=6)	N/A	1536	2.00E-04	1,024
Perceiver AR	62M (D=768, L=6)	N/A	256	2.00E-04	8,192 (768 latents)
MEGABYTE	125M (D=768, L=12)	125M (D=768, L=12)	16	2.00E-04	196,608 (patch size 192)
w/o local model	2.7B (D=4096, L=32)	N/A	16	2.00E-04	196,608 (patch size 48)
w/o global model	125M (D=768, L=12)	125M (D=768, L=12)	16	2.00E-04	196,608 (patch size 192)
w/o cross-patch Local model	250M	156M (D=768, L=15)	16	2.00E-04	196,608 (patch size 192)
w/ CNN encoder	125M (D=768, L=12)	125M (D=768, L=12)	16	2.00E-04	196,608 (patch size 192)
Image task 640					
Transformer	83M (D=768, L=8)	N/A	4800	3.00E-04	1,024
Perceiver AR	62M (D=768, L=6)	N/A	2048	3.00E-04	4,096 (1024 latents)
MEGABYTE	125M (D=768, L=12)	83M (D=768, L=8)	32	3.00E-04	1,228,800 (192 patch size)
audio					
Transformer	135M (D=768, L=13)	N/A	2048	2.00E-04	1024
Perceiver AR	62M (D=768, L=6)	N/A	384	2.00E-04	8,192 (1024 latents)
MEGABYTE	350M (D=1024, L=24)	125M (D=768, L=12)	256	2.00E-04	524,288 (32 patch size)
w/o local model	2.7B (D=4096, L=32)	125M (D=768, L=12)	256	2.00E-04	524,288 (32 patch size)
w/o global model	350M (D=1024, L=24)	125M (D=768, L=12)	256	2.00E-04	524,288 (32 patch size)
w/o cross-patch Local model	350M (D=1024, L=24)	146M (D=768, L=14)	256	2.00E-04	524,288 (32 patch size)
w/ CNN encoder	350M (D=1024, L=24)	125M (D=768, L=12)	256	2.00E-04	524,288 (32 patch size)

Table 9: **Model architecture details.** We report the model size, the embedding size (D), number of layaers(L), total batch size (BS), learning rate(LR), and context length. When we vary the number of model layers from the standard amount for the given size (Table 8), we note this accordingly. For PerceiverAR models, we note the number of latents used, and for MEGABYTE models we note the patch sizes.

B Pseudocode

Listing 1: Pseudocode of Megabyte model

```
class MegaByteDecoder:
    def __init__(
        self,
        global_args,
        local_args,
        patch_size,
    ):
        self.pad = 0
        self.patch_size = patch_size
        self.globalmodel = TransformerDecoder(global_args)
        self.localmodel = TransformerDecoder(local_args)

    def forward(
        self,
        bytes,
    ):
        bytes_global, bytes_local = self.prepare_input(bytes)
```

```
        global_bytes_embedded = self.globalmodel.embed(bytes_global)
        global_in = rearrange(
            global_bytes_embedded,
            "b (t p) e -> b t (p e)",
            p=self.patch_size,
        )
        global_output = self.globalmodel(global_in)

        global_output_reshaped = rearrange(
            global_output,
            "b t (p e) -> (b t) p e",
            p=self.patch_size,
        )
        local_bytes_embedded = self.localmodel.embed(bytes_local)
        local_in = local_bytes_embedded + global_output_reshaped
        local_output = self.localmodel(local_in)

        batch_size = bytes_global.shape[0]
        x = rearrange(local_output, "(b t) l v  -> b (t l) v", b=
            batch_size)
        return x

    def prepare_input(self, bytes):
        padding_global = bytes.new(bytes.shape[0], self.patch_size).
            fill_(self.pad)
        bytes_global = torch.cat((padding_global, bytes[:, : -self.
            patch_size]), -1)

        bytes_input = rearrange(bytes, "b (t p) -> (b t) p", p=self.
            patch_size)
        padding_local = bytes_input.new(bytes_input.shape[0], 1).fill_
            (self.pad)
        bytes_local = torch.cat((padding_local, bytes_input[:, :-1]),
            -1)

        return bytes_global, bytes_local
```

C PerceiverAR Implementation

To reproduce PerceiverAR in a compute-controlled setting we extended the standard transformer implementation in metaseq with an additonal cross attention layer to compute the latents and match the architecture of PerceiverAR. We trained the model by sampling random spans from each text, matching the procedure used in the PerceiverAR codebase. To be consistent with the original work, we use sliding window evaluation with a stride of $num\_latents/2$ unless otherwise noted. In several cases we used the standard metaseq implementation as opposed to specific techniques reported in the original paper: 1) we used standard attention dropout instead of cross-attention dropout 2) We did not implement chunked attention. We verified our implementation by reproducing the "Standard Ordering" experiments in Table 5 of the Perceiver AR paper. After carefully matching context size, number of latents, the amount of data and training steps used and learning rate, we achieved 3.53 bpb vs 3.54 reported in the original paper.

D More results

D.1 Patch scan Implementation

Images have a natural structure, containing a grid of $n \times n$ pixels each composed of 3 bytes (corresponding to color channels). We explore two ways of converting images to sequences for modeling (see Figure 8). Firstly, *raster scan* where the pixels are linearized into 3 bytes and concatenated row-by-row. Secondly, *patch scan* where we create patches of shape $p \times p \times 3$ bytes

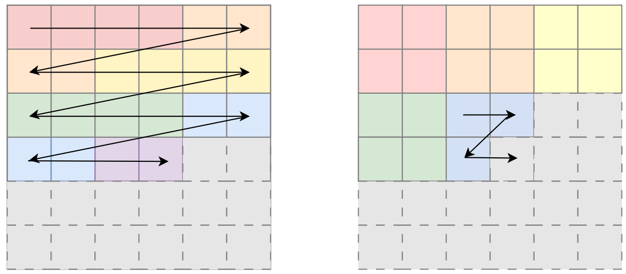

Figure 8: Two ways to model 2D data sequentially. Left, raster scan, by taking bytes row by row and left to right; right, patch scan, where we first split an image into patches, and do raster scan across patches and within a patch. (T=36, K=9, P=4).

where $p = \sqrt{\frac{P}{3}}$, and then use a raster scan both within and between patches. Unless otherwise specified, MEGABYTE models use *patch scan* for image data.

D.2 Patch scan vs Raster scan

The patch scan method is inspired by recent works in Vision Transformers (Dosovitskiy et al., 2020), and it is more effective than raster scan for modeling image sequencing. We found it improves both MEGABYTE and Perceiver AR.

	(Global) Size	Local Size	context	bpb
MEGABYTE (patch scan)	62M (D=768, L=6)	N/A	8,192 (768 latents)	3.158
MEGABYTE (raster scan)	62M (D=768, L=6)	N/A	8,192 (768 latents)	3.428
Perceiver AR (patch scan)	125M (D=768, L=12)	125M (D=768, L=12)	196,608 (patch size 192)	3.373
Perceiver AR (raster scan)	125M (D=768, L=12)	125M (D=768, L=12)	196,608 (patch size 192)	3.552

Table 10: ImageNet256 performance with patch scan vs raster scan for MEGABYTE and Perceiver AR.

D.3 Longer sequence modeling

For our pg19 scaling experiment, we also use longer context length for MEGABYTE. The results are shown in Table 11. With longer sequence, we didn't observer further improvement, consistent with findings in Hawthorne et al. (2022). We think we will benefit more from longer sequence when we futher scale up the model size and data.

	context	bpb
MEGABYTE	8,192 (patch size 8)	0.8751
MEGABYTE	16,384 (patch size 8)	0.8787

Table 11: Longer sequence for PG19 dataset. For both experiments, we set global model as 1.3b, local model as 350m, and MEGABYTE patch size as 8.