
Supplementary Material of SHF: Symmetrical Hierarchical Forest with Pretrained Vision Transformer Encoder for High-Resolution Medical Segmentation

Anonymous Author(s)

Affiliation

Address

email

1 Discussions and Ablations

1.1 Sequence length L and compressed ratio γ .

The impact of various sequence lengths on the compression rate of the original image can be different when using the same input image and edge extraction algorithm. The first column shows an edge image at a resolution of 1024×1024 pixels. We tested sequence lengths L of [256, 1024, 2050], corresponding to average patch sizes of [31.7, 4.3, 9.17] pixels, which are required for the grid patch configuration. The resulting compression rates γ are [4.07, 12.18, 27.66]. For image segmentation, a sequence length of 1024 appears optimal for images with a $1K$ resolution. Notably, the compression rates [6.06, 22.47, 47.76] are higher in 3D, where increased dimensions lead to sparser single patch information, making experiments in higher visual dimensions feasible.

The core reason why SHF can handle small patch sizes at high resolutions is that the sequence size can be reduced with hierarchical forest patching. The extent to which the sequence length can be reduced by adjusting the split value of the HierarchicalForest, without significantly losing prediction performance. The split value v controls the total length and distribution of patch sizes. When the split value is halved [100, 50, 20], the patch size distribution or the average patch size [12.77, 19.17, 29.88] is also close to being halved. This means the average patch size grows linearly with the split value. For the uniform grid patching strategy, the sequence length grows by $O(\frac{L}{P})^2$. However, we observed an approximately linear increase in the average sequence length as the average patch size decreased. Notably, the average patch size [6.27, 11.79, 17.86] are higher in 3D, where increased dimensions lead to sparser single patch information, making experiments in higher visual dimensions feasible.

1.2 Information lost of patching and depatching.

We present the hierarchical forest structure generated from the input image using the patch partitioning strategy, showing the effects of compressing and reconstructing the original mask. When decoding the mask through spatial partitioning, the accuracy can theoretically reach an upper limit of 99.5% with perfect predictions. However, as our experiment's sequence mask achieves only 82.97%, this forest-structured reconstruction's impact on decoding performance is relatively minor.

Notably, SHF differs from a convolution-based decoder in terms of image quality degradation. We believe this is due to two types of losses: the first is texture loss, where incorrect model regression generates inaccurate textures, leading to noise artifacts in the SHF image. The second is structural loss, observed as an amplification of noise in patches with varying structures introduced during the Dispatching stage. We attribute both losses to the loss of geometric relationships between patches during compression. Figure 1 illustrates the degradation of these geometric properties.

33 1.3 Hyper-parameters of Hierarchical Forest.

34 We show the effects of hyperparameters, including Gaussian smooth kernel size K and sequence
 35 length L , on image encoding. We set the Gaussian smooth kernel to $[1, 3, 5, 7]$ at 512 resolution. We
 36 can see that a larger kernel corresponds to a smoother edge image. We set the sequence length of
 37 hierarchical forest patching to $[512, 384, 256, 128]$ at 512 resolution. We can see that with different
 38 Gaussian smooth kernels combined with sequence length, we can extract image feature information
 39 at different levels. We show codes for building a hierarchical Forest, including a patchify method:

```

40
41 class Patchify(torch.nn.Module):
42     def __init__(self, sths=[1,3,5,7], fixed_length=1024, \
43         cannys=[50, 100], patch_size=8, shift=50) -> None:
44         super().__init__()
45
46         self.sths = sths
47         self.fixed_length = fixed_length
48         self.cannys = [x for x in range(cannys[0], cannys[1], 1)]
49         self.patch_size = patch_size
50
51     def forward(self, img, target): # we assume inputs are always structured like this
52         # Do some transformations. Here, we're just passing though the input
53
54         self.smooth_factor = random.choice(self.sths)
55         c = random.choice(self.cannys)
56         self.canny = [c, c+shift]
57
58         grey_img = cv.GaussianBlur(img, (self.smooth_factor, self.smooth_factor), 0)
59         edges = cv.Canny(grey_img, self.canny[0], self.canny[1])
60         qdt = FixedHierarchical(domain=edges, fixed_length=self.fixed_length)
61         seq_img = qdt.serialize(img, size=(self.patch_size, self.patch_size, 3))
62         seq_img = np.asarray(seq_img)
63         seq_img = np.reshape(seq_img, [self.patch_size, -1, 3])
64
65         seq_mask = qdt.serialize(target, size=(self.patch_size, self.patch_size, 1))
66         seq_mask = np.asarray(seq_mask)
67         seq_mask = np.reshape(seq_mask, [self.patch_size, -1, 1])
68
69         return seq_img, seq_mask, qdt

```

70 1.4 Training: Algorithm, Hyper-parameters & Loss function

71 Since depatching happens in the evaluate stage, our algorithm is also divided into the training stage,
 72 and the evaluate stage. Let's first look at the training stage, where L represents the length of the
 73 hierarchical patching sequence, K represents the optional kernel size, t_l and t_h represent the lower and
 74 higher threshold of canny edge detection, f is the model, x is the input, θ is the trainable parameter,
 75 D is the dataset, N is the number of batches, and E is the total number of epochs required.

76 As shown in the training stage of overview, the input image x first goes through Gaussian smoothing
 77 and canny edge detection to get x_e , then the edge image x_e goes through hierarchical patching to
 78 get the HierarchicalForest T_x and an encoded sequence of length L , and we also use T_x to encode
 79 the mask y to get y_p . Then, we use the model to train on data with x_p as input and y_p as mask.
 80 After obtaining the predicted encoded sequence mask \hat{y}_p for the evaluation stage, we send it to the
 81 hierarchical dispatching stage and reconstruct the prediction \hat{y} . Then, we can calculate the dice score
 82 between ground truth y and \hat{y} .

83 We train the model using the AdamW optimizer with an initial learning rate of $1e-4$ over 800 epochs.
 84 The first 20 epochs are allocated for learning rate warm-up, followed by a decay by a factor of 0.1 at
 85 epochs 400 and 600. By epoch 800, the model converges. To maximize training speed, we select the

largest possible batch size possible within the GPU memory limits. The loss function combines Dice loss and cross-entropy (BCE) loss:

```

class MulticlassDiceLoss(torch.nn.Module):
    def __init__(self, smooth=1e-6):
        super(MulticlassDiceLoss, self).__init__()
        self.smooth = smooth

    def forward(self, pred, target):
        # pred: (N, C, H, W) - raw logits
        # target: (N, C, H, W) - one-hot encoded

        # Apply softmax to get probabilities
        pred = torch.softmax(pred, dim=1)

        # Calculate dice per channel
        intersection = torch.sum(pred * target, dim=(2, 3))
        union = torch.sum(pred + target, dim=(2, 3))

        dice = (2. * intersection + self.smooth) / (union + self.smooth)

        # Average across channels
        loss = 1. - dice.mean()

        return loss

class DiceCELoss(nn.Module):
    """Combination of CrossEntropy and Dice loss"""
    def __init__(self, weight_ce=1.0, weight_dice=1.0):
        super(DiceCELoss, self).__init__()
        self.ce = nn.CrossEntropyLoss()
        self.dice = MulticlassDiceLoss()
        self.weight_ce = weight_ce
        self.weight_dice = weight_dice

    def forward(self, inputs, targets):
        ce_loss = self.ce(inputs, targets)
        dice_loss = self.dice(inputs, targets)
        total_loss = self.weight_ce * ce_loss + self.weight_dice * dice_loss
        return total_loss

```

The parameter w controls the balance between CE loss and dice loss, set to 0.5 in our experiments. To stabilize calculations, the smoothing term ϵ is set to 1.0.

1.5 Learning From the Encoded Image Space

A key concern is to ensure is that the spatial structure learned in the embeddings \hat{y} corresponds to the spatial structure of the sequence s . We can assume that if the output prediction \hat{y} can completely match the compressed mask y_c , then there should be a minimal loss in translation to the original mask through the spatial structure. To achieve this, we expect the regression ability of the transformer to learn the implicit spatial matching from the embedding space to the geometric pixel space, as empirically observed by a direct decrease in the loss. Since, for the positional encoding, we chose a trainable embedding vector with an initial value of zero, we speculate that if the order of patches, after hierarchical forest patching, affects the learning, then the trainable positional encoding may start to learn the positioning of tokens as manifested by the Z-order curve of the forest.

1.6 Experimental Setup

All the experiments were performed using the Frontier Supercomputer at ORNL. Each Frontier node has a single 64-core AMD EPYC CPU and four AMD Instinct MI250X GPUs (128GB per GPU).

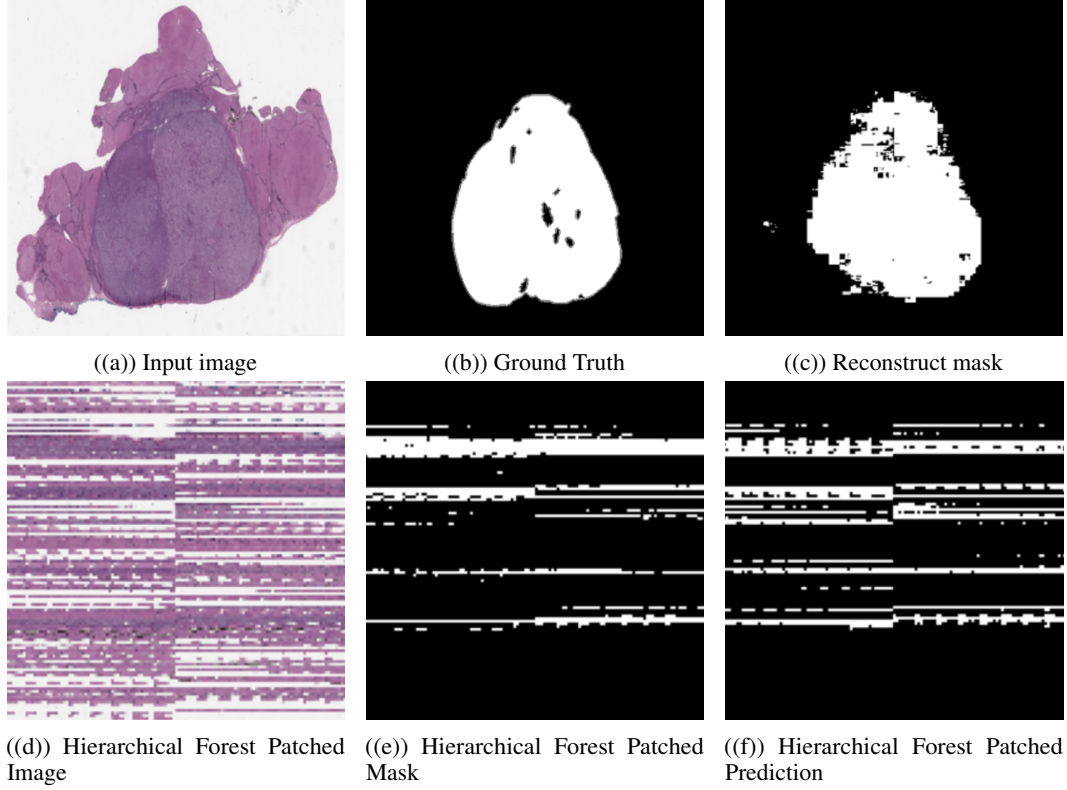


Figure 1: This figure shows an example of an hierarchical forest patching image and reconstructed mask from the patching mask. The geometric relationship between patches is lost when compressing from the image domain to the Sequence domain. Thus, transformers need to learn the non-geometry sequence without the assumption of spatial information.

140 The four MI250X GPUs are connected with Infinity Fabric GPU-GPU of 50GB/s. The nodes are
 141 connected via a Slingshot-11 interconnect with 100GB/s, to a total of 9,408 nodes. For the software
 142 stack, we used Pytorch 2.4 nightly build 03/16/2024. ROCm v5.7.0, MIOpen v2.19.0, RCCL v2.13.4
 143 with libfabric v1.15.2 plugin.

144 1.7 Codes for SHF

145 Codes for building a recursive forest function, including the depatching stage:

```

146 class Rect:
147     def __init__(self, x1, x2, y1, y2) -> None:
148         # *q
149         # p*
150         self.x1 = x1
151         self.x2 = x2
152         self.y1 = y1
153         self.y2 = y2
154
155         assert x1<=x2, 'x1 > x2, wrong coordinate.'
156         assert y1<=y2, 'y1 > y2, wrong coordinate.'
157
158     def contains(self, domain):
159         patch = domain[self.y1:self.y2, self.x1:self.x2]
160         return int(np.sum(patch)/255)
161
162     def get_area(self, img):

```

```

163         return img[self.y1:self.y2, self.x1:self.x2, :]
164
165     def set_area(self, mask, patch):
166
167         patch_size = self.get_size()
168         # patch = np.resize(patch, patch_size)
169         patch = patch.astype('float32')
170         patch = cv.resize(patch, interpolation=cv.INTER_CUBIC , dsize=patch_size)
171         if len(patch.shape)==2:
172             patch = np.expand_dims(patch, axis=-1)
173
174         mask[self.y1:self.y2, self.x1:self.x2, :] = patch
175         return mask
176
177     def get_coord(self):
178         return self.x1,self.x2,self.y1,self.y2
179
180     def get_size(self):
181         return self.x2-self.x1, self.y2-self.y1
182
183     def get_center(self):
184         return (self.x2+self.x1)/2, (self.y2+self.y1)/2
185
186     def draw(self, ax, c='grey', lw=0.5, **kwargs):
187         # Create a Rectangle patch
188         import matplotlib.patches as patches
189         rect = patches.Rectangle((self.x1, self.y1),
190                                 width=self.x2-self.x1,
191                                 height=self.y2-self.y1,
192                                 linewidth=lw, edgecolor='w', facecolor='none')
193         ax.add_patch(rect)
194
195     def draw_area(self, ax, c='green', lw=0.5, **kwargs):
196         # Create a Rectangle patch
197         import matplotlib.patches as patches
198         rect = patches.Rectangle((self.x1, self.y1),
199                                 width=self.x2-self.x1,
200                                 height=self.y2-self.y1,
201                                 linewidth=lw, edgecolor='w', facecolor=c)
202         ax.add_patch(rect)
203
204     def draw_rescale(self, ax, c='green', lw=0.5, **kwargs):
205         # Create a Rectangle patch
206         import matplotlib.patches as patches
207         rect = patches.Rectangle((self.x1, self.y1),
208                                 width=16,
209                                 height=16,
210                                 linewidth=lw, edgecolor='w', facecolor=c)
211         ax.add_patch(rect)
212
213     def draw_zorder(self, ax, c='red', lw=0.5, **kwargs):
214         # Create a Rectangle patch
215         import matplotlib.patches as patches
216         rect = patches.Rectangle((self.x1, self.y1),
217                                 width=16,
218                                 height=16,
219                                 linewidth=lw, edgecolor='w', facecolor=c)
220         ax.add_patch(rect)
221

```

222

```

223 class FixedHierarchicalForest:
224     def __init__(self, domain, fixed_length=128, build_from_info=False, meta_info=None) -> None:
225         self.domain = domain
226         self.fixed_length = fixed_length
227         if build_from_info:
228             self.nodes = self.decoder_nodes(meta_info=meta_info)
229         else:
230             self._build_hierarchical_forest()
231
232     def nodes_value(self):
233         meta_value = []
234         for rect,v in self.nodes:
235             size,_ = rect.get_size()
236             meta_value += [[size/8]]
237         return meta_value
238
239     def encode_nodes(self):
240         meta_info = []
241         for rect,v in self.nodes:
242             meta_info += [[rect.x1,rect.x2,rect.y1,rect.y2]]
243         return meta_info
244
245     def decoder_nodes(self, meta_info):
246         nodes = []
247         for info in meta_info:
248             x1,x2,y1,y2 = info
249             n = Rect(x1, x2, y1, y2)
250             v = n.contains(self.domain)
251             nodes += [[n,v]]
252         return nodes
253
254     def _build_hierarchical_forest(self):
255
256         h,w = self.domain.shape
257         assert h>0 and w >0, "Wrong img size."
258         root = Rect(0,w,0,h)
259         self.nodes = [[root, root.contains(self.domain)]]
260         while len(self.nodes)<self.fixed_length:
261             bbox, value = max(self.nodes, key=lambda x:x[1])
262             idx = self.nodes.index([bbox, value])
263             if bbox.get_size()[0] == 2:
264                 break
265
266             x1,x2,y1,y2 = bbox.get_coord()
267             lt = Rect(x1, int((x1+x2)/2), int((y1+y2)/2), y2)
268             v1 = lt.contains(self.domain)
269             rt = Rect(int((x1+x2)/2), x2, int((y1+y2)/2), y2)
270             v2 = rt.contains(self.domain)
271             lb = Rect(x1, int((x1+x2)/2), y1, int((y1+y2)/2))
272             v3 = lb.contains(self.domain)
273             rb = Rect(int((x1+x2)/2), x2, y1, int((y1+y2)/2))
274             v4 = rb.contains(self.domain)
275
276             self.nodes = self.nodes[:idx] + [[lt,v1], [rt,v2], [lb,v3], [rb,v4]] \
277                 + self.nodes[idx+1:]
278
279             # print([v for _,v in self.nodes])

```

```

280
281 def count_patches(self):
282     return len(self.nodes)
283
284 def serialize(self, img, size=(8,8,3)):
285
286     seq_patch = []
287     seq_size = []
288     seq_pos = []
289     for bbox,value in self.nodes:
290         seq_patch.append(bbox.get_area(img))
291         seq_size.append(bbox.get_size()[0])
292         seq_pos.append(bbox.get_center())
293
294     h2,w2,c2 = size
295
296     for i in range(len(seq_patch)):
297         h1, w1, c1 = seq_patch[i].shape
298         assert h1==w1, "Need squared input."
299         seq_patch[i] = cv.resize(seq_patch[i], (h2, w2), interpolation=cv.INTER_NEAREST)
300         # assert seq_patch[i].shape == (h2,w2,c2), "Wrong shape {} get, need {}".format(seq_patch[i].shape, (h2,w2,c2))
301     if len(seq_patch)<self.fixed_length:
302         # import pdb
303         # pdb.set_trace()
304         if c2 > 1:
305             seq_patch += [np.zeros(shape=(h2,w2,c2))] * (self.fixed_length-len(seq_patch))
306         else:
307             seq_patch += [np.zeros(shape=(h2,w2))] * (self.fixed_length-len(seq_patch))
308             seq_size += [0]*(self.fixed_length-len(seq_size))
309             seq_pos += [tuple([-1,-1])]*(self.fixed_length-len(seq_pos))
310     elif len(seq_patch)>self.fixed_length:
311         pass
312         # random_drop
313     assert len(seq_patch)==self.fixed_length, "Not equal fixed length."
314     assert len(seq_size)==self.fixed_length, "Not equal fixed length."
315     return seq_patch, seq_size, seq_pos
316
317 def deserialize(self, seq, patch_size, channel):
318
319     H,W = self.domain.shape
320     seq = np.reshape(seq, (self.fixed_length, patch_size, patch_size, channel))
321     seq = seq.astype(int)
322     mask = np.zeros(shape=(H, W, channel))
323     print("demask:", mask.shape)
324
325     # import pdb;pdb.set_trace()
326     # mask = np.expand_dims(mask, axis=-1)
327     for idx,(bbox,value) in enumerate(self.nodes):
328         pred_mask = seq[idx, ...]
329         mask = bbox.set_area(mask, pred_mask)
330     return mask
331
332 def draw(self, ax, c='grey', lw=1):
333     for bbox,value in self.nodes:
334         bbox.draw(ax=ax)
335
336 def draw_area(self, ax, c='green', lw=1):
337     for bbox,value in self.nodes:
338         bbox.draw_area(ax=ax, c=c, lw=lw)

```

```

339
340 def draw_rescale(self, ax, c='green', lw=1):
341     for bbox,value in self.nodes:
342         bbox.draw_rescale(ax=ax, c=c, lw=lw)
343
344 def draw_zorder(self, ax, c='red', lw=1):
345     xs = []
346     ys = []
347     for bbox,value in self.nodes:
348         x,y = bbox.get_center()
349         xs += [x]
350         ys += [y]
351     ax.plot(xs, ys, color='red', linewidth=1)
352
353 Training codes for SHF:
354
355     train_sampler = torch.utils.data.distributed.DistributedSampler(train_set)
356     val_sampler = torch.utils.data.distributed.DistributedSampler(val_set, shuffle=False)
357
358     train_loader = DataLoader(train_set, batch_size=args.batch_size, sampler=train_sampler, colla
359     val_loader = DataLoader(val_set, batch_size=args.batch_size, sampler=val_sampler, collate_fr
360     test_loader = val_loader
361
362 # Training loop
363 num_epochs = args.epoch
364 train_losses = []
365 val_losses = []
366 output_dir = args.savefile # Change this to the desired directory
367 os.makedirs(output_dir, exist_ok=True)
368 import time
369 for epoch in range(num_epochs):
370     model.train()
371     epoch_train_loss = 0.0
372     train_loader.sampler.set_epoch(epoch)
373     start_time = time.time()
374     step=1
375     for batch in train_loader:
376         # with torch.autocast(device_type='cuda', dtype=torch.float16):
377         image, mask, qimages, qmasks, qdt, seq_size, seq_pos = batch
378         qimages, qmasks = qimages.to(device_id), qmasks.to(device_id)
379         seq_size, seq_pos = seq_size.to(device_id), seq_pos.to(device_id)
380         seq_size= seq_size.unsqueeze(-1)
381         seq_ps = torch.concat([seq_size, seq_pos],dim=-1)
382
383         optimizer.zero_grad()
384         outputs = model(qimages, seq_ps)
385         loss = criterion(outputs, qmasks)
386         score = dice_score(outputs, qmasks)
387
388         loss.backward()
389         optimizer.step()
390         epoch_train_loss += loss.item()
391         step+=1
392     end_time = time.time()
393
394 with torch.no_grad():
395     if (epoch - 1) % 10 == 9 and device_id == 0:
396         sub_trans_plot(image, mask, qmasks=qmasks, pred=outputs, qdt=qdt,
397             fixed_length=args.fixed_length, bi=-1, epoch=epoch, output_dir=args.savefile)

```



```

397         epoch_train_loss /= len(train_loader)
398         train_losses.append(epoch_train_loss)
399         scheduler.step()

400 Scripts for job launching resolution 8192 with fixed length 10,201 on 8 nodes (64 GPUs) with SHF:

401 source export_ddp_envs.sh
402
403 module load PrgEnv-gnu
404 module load gcc/12.2.0
405 module load rocm/5.7.0
406
407 # exec
408 srun -N 8 -n 64 --ntasks-per-node 8 python ./train/sam_trans_ddp.py \
409     --data_dir=./paip/output_images_and_masks \
410     --resolution=8192 \
411     --fixed_length=10201 \
412     --patch_size=8 \
413     --pretrain=sam-b \
414     --epoch=1000 \
415     --batch_size=1 \
416     --savefile=./sam-b-trans-res8k-f10k-pz8-n8

```