A APPENDIX

A.1 DETAILS OF EXPERIMENTAL SETTINGS

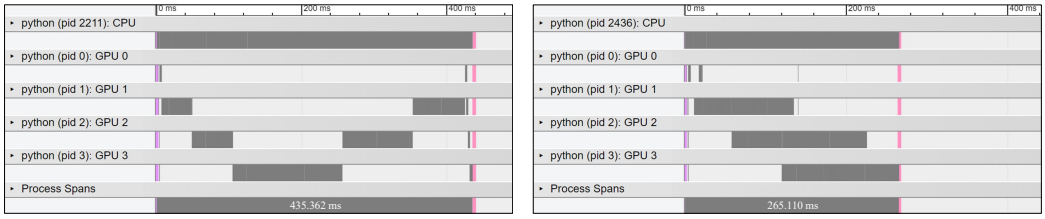
In our vision tasks, both BP and SCPL adopt a cosine learning rate scheduler, starting from an initial learning rate of 10e-3 and decaying to 10e-5. We use Adam as the optimizer. For data augmentation, we refer to the settings in Khosla et al. (2020) on CIFAR-10 and CIFAR-100, where each image undergoes resizing, random cropping, random horizontal flipping, jittering, and random grayscaling to generate two augmented views as inputs. Consequently, the batch size increases from the original N to $2N$. Regarding the batch size, in the accuracy experiments, BP is set to 128, while SCPL is set to 1024. However, in training time measurement experiments, both are tested under 32, 64, 128, 256, and 512. The training epochs are set to 200. In the SCPL configuration, all models use an MLP (multi-layer perceptron) as the projection head, with a structure of $Linear(dim, 512) - ReLU() - Linear(512, 1024)$. Here, dim represents the dimension after flattening the feature map. The temperature parameter τ is set to 0.1 for all models. Furthermore, in the training time experiments, each component is placed on a separate GPU. The detailed configurations for VGG and ResNet are as follows.

- VGG: It consists of 4 max-pooling layers (MP) and 6 convolutional layers (Conv). Each convolutional layer uses ReLU as the activation function and employs batch normalization (BN). The classifier consists of 2 fully connected layers (FC) with sigmoid as the activation function between the two layers. In our implementation, SCPL splits VGG into 4 components, structured as $component_1 - component_2 - component_3 - component_4$. The $component_1$ and $component_2$ are composed of $[[Conv - BN - ReLU] \times 2 - MP]$. The $component_3$ and $component_4$ are composed of $[[Conv - BN - ReLU] - MP]$. However, an additional classifier is included in $component_4$, resulting in $[component_4 - [FC - sigmoid - FC]]$.
- ResNet: It is an 18-layer residual neural network (ResNet-18) with a linear fully connected layer as the classifier. In our implementation, SCPL splits ResNet into 4 components, structured as $component_1 - component_2 - component_3 - component_4$. The $component_1$ is structured as $[StemBlock - BasicBlock \times 2]$. The $component_2$, $component_3$, and $component_4$ are structured as $[BasicBlock \times 2]$. However, an additional classifier is included in the last $component_4$, resulting in $[component_4 - [FC]]$. The $StemBlock$ includes $[Conv - BN - ReLU]$. The $StemBlock$ is composed of a convolutional layer followed by batch normalization (BN) and Rectified Linear Unit (ReLU) activation. The $BasicBlock$ is constructed with a convolutional layer using the LeakyReLU activation function, another convolutional layer, and a skip connection that adds a fully connected transformation to the input of the $BasicBlock$ module. Finally, the output passes through another LeakyReLU activation.

In NLP tasks, both BP and SCPL use a fixed learning rate of 10e-3 and employ Adam as the optimizer. All texts in the datasets undergo preprocessing steps such as creating word indices, removing stop words, and limiting the maximum text word length T, which is a hyperparameter representing the sentence length for each sample. Data augmentation is not utilized, and therefore, the batch size remains at its original value N. For the AG's news dataset, the maximum text word length per sample is set to 60, while for IMDB, it is set to 350. The training epochs for both BP and SCPL models are set to 50. Regarding the batch size, we experimented with 16, 32, 64, 128, 256, 384, 512, 768, 1024, 1280, 1536, 1792, 2048, and 4096, and we present the results with the best accuracy in this paper. Additionally, both BP and SCPL models utilize pre-trained Glove word embeddings (Pennington et al., 2014) of dimensionality 300 in the first layer of the model. In the configuration of SCPL, all models by default use an identity function, $f(x) = x$, as the projection head in training. The temperature parameter τ is set to 0.1 for all models.

Detailed architectures of LSTM and Transformer are as follows.

- LSTM: It consists of 3 bi-LSTM hidden layers (each with a dimensionality of 300) and 1 Glove embedding layer at the beginning of the model. At the end of the model, there are 2 fully connected layers serving as the classifier. The Tanh function is used as the activation function between the two layers. SCPL splits the LSTM model into 4 components, structured as $component_1 - component_2 - component_3 - component_4$. $Component_1$ represents



(a) Training LSTM on IMDB (using NMP). (b) Training LSTM on IMDB (using SCPL).

Figure 4: Visualizing the training job of each device.

the $[GloveEmb]$ layer, while $component_2$, $component_3$, and $component_4$ represent the $[LSTM]$ layers. However, an additional classifier is included in $component_4$, resulting in $[component_4 - [FC - tanh - FC]]$.

- Transformer: It consists of 3 Transformer encoders (each with a dimensionality of 300 and a dropout rate of 0.1) and 1 Glove embedding layer at the beginning of the model. At the end of the model, there are 2 fully connected layers serving as the classifier. The Tanh function is used as the activation function between the two layers. SCPL splits the Transformer model into 4 components, structured as $component_1 - component_2 - component_3 - component_4$. $Component_1$ represents the $[GloveEmb]$ layer, while $component_2$, $component_3$, and $component_4$ represent the $[Transformer]$ layers. However, an additional classifier is included in $component_4$, resulting in $[component_4 - [FC - tanh - FC]]$.

A.2 Profiling NMP and SCPL

We used PyTorch's profiler to observe the operating periods of the CPU and the GPUs of one iteration. We used 4 GPUs to train an LSTM with 4 layers; each GPU is responsible for the training of one layer.

Figure 4 shows the CPU's working periods and each GPU's working periods when training by NMP and SCPL. The top row shows the CPU's running periods. Since the CPU handles task scheduling, data preprocessing, data management, and some non-parallel computation, the CPU is running throughout the training periods.

When training by NMP (Figure 4(a)), the GPU0 performs forward for layer 1, then GPU1 performs forward for layer 2, then GPU2 performs forward for layer 3, then GPU3 performs forward for layer 4. GPU3 continues to perform backward for layer 4, then GPU2 continues to perform backward for layer 3, then GPU1 continues to perform backward for layer 2, then GPU0 continues to perform backward for layer 1. Finally, the CPU asks all GPUs to update the parameters based on the computed gradients (the red bars). As shown, all the GPUs perform operations sequentially, causing backward locking, so many bubbles exist among the dependent tasks. The total training time for this iteration is 435.362 ms.

When training by SCPL (Figure 4(b)), the operations on different GPUs may overlap. In particular, when GPU0 finishes the forward for layer 1, the following operations may occur simultaneously: backward for layer 1 (on GPU0) and forward for layer 2 (on GPU1). Similarly, when GPU1 finishes the forward for layer 2, backward for layer 2 (on GPU1) and forward for layer 3 (on GPU2) may occur simultaneously. After GPU2 finishes the forward, backward for layer 3 (on GPU2) and forward for layer 4 (on GPU3) may occur concurrently. Finally, GPU3 performs the backward for layer 4, and then the CPU issues an update command for all GPUs (the red bars). Since many bubbles are removed, the total training time for this iteration is reduced to 265.110 ms.

A.3 More comparisons on empirical training time

This section shows the empirical training time per epoch for NLP and vision tasks using famous network architectures.

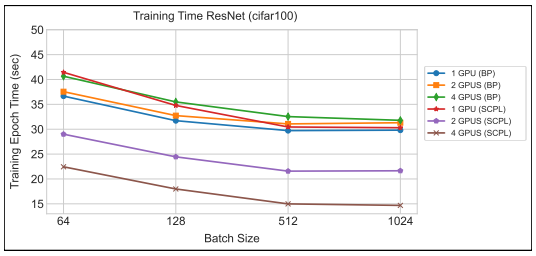


Figure 5: Empirical training time per epoch using ResNet architecture on CIFAR-100.

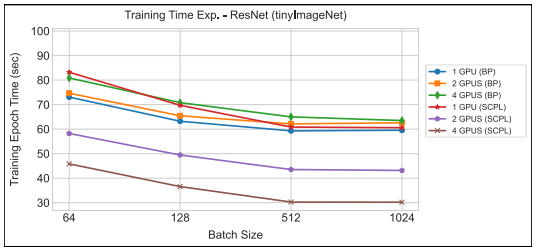


Figure 6: Empirical training time per epoch using ResNet architecture on tiny-ImageNet.

For the NLP tasks, we apply the LSTM and Transformer networks; the experimental datasets include AGNews and IMDB. Regarding vision tasks, we select the VGG network and ResNet, and the dataset includes CIFAR-100 and tiny-ImageNet.

The results are shown in Figures 5, 6, 8, 8, 9, 10, 11, and 12. When using 4 GPUs, SCPL is approximately 2 times faster than BP on vision tasks and approximately 1.6 times faster on NLP tasks.

A.4 More comparisons on test accuracies

Table 6: A comparison of the test accuracies of different methodologies when using different neural network architectures on AG's news. We follow the same notations used in Table 4.

	LSTM	Transformer
BP	91.97 ± 0.19	91.27 ± 0.18
Early Exit	85.91 ± 0.11	85.79 ± 0.43
AL	91.53 ± 0.20	$\mathbf{91.17} \pm 0.43$
SCPL	$\mathbf{92.12} \pm 0.04$ †	$\mathbf{91.64} \pm 0.23$ †

This section shows the test accuracies for NLP and vision tasks using famous network architectures.

Figure 6 shows the test accuracies of LSTM and Transformer on AG's news when these models are trained by BP, Early Exit, AL, and SCPL. We report the mean and standard deviation of 5 trials.

Similarly, we also report the results on CIFAR-10 and CIFAR-100, using the vanilla convolutional neural network (Vanilla ConvNet), VGG, and ResNet as the network structures. The results are shown in Table 7 and Table 8.

In general, SCPL consistently outperforms other local-objective-based learning strategies in the experimented datasets and different network architectures.

A.5 SCPL vs. GPipe

SCPL and GPipe share an architectural similarity: they both rely on pipelining to realize model parallelism and enhance the training throughput. However, they adopt distinct strategies to address

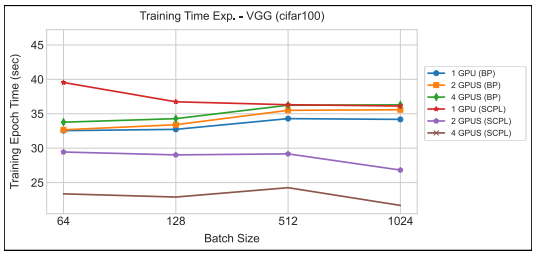


Figure 7: Empirical training time per epoch using VGG architecture on CIFAR-100.

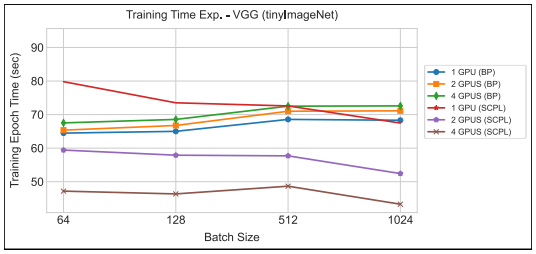


Figure 8: Empirical training time per epoch using VGG architecture on tiny-ImageNet.

the challenges posed by forward and backward locking. SCPL and GPipe can be integrated to further improve training throughput.

SCPL focuses on mitigating backward locking, where the sequential dependency of gradient calculations across layers impedes parallelism. SCPL introduces a local objective for each component. These local objectives serve to disentangle the gradient computation process, allowing greater concurrency and minimizing the impact of backward locking. Referring to Figure 3 and the top subfigure in Figure 13, the backward pass in different components can be computed simultaneously in different GPUs.

GPipe tackles forward locking, a phenomenon in which the forward operation of a layer must wait for the completion of the forward operations in the earlier layers. GPipe alleviates the constraint by subdividing traditional mini-batches into micro-batches, allowing for an overlap of computations between the forward passes of different layers. This approach mitigates the impact of forward locking. Referring to the middle subfigure in Figure 13, each mini-batch is further divided into 3 micro-batches. Particularly, letting FWℓ refer to the forward operations of a mini-batch at layer ℓ, we use F^1_ℓ, F^2_ℓ, and F^3_ℓ to refer to the forward pass of the three micro-batches in this layer. In this setting, once a GPUi finishes the computation of F^1_ℓ, the GPU$(i+1)$ can continue to execute $F^1_{\ell+1}$, and the GPUi operates F^2_ℓ simultaneously. As a result, it is possible to execute the forward passes at different layers simultaneously.

Given their complementary strengths in addressing forward and backward locking, it is possible to integrate SCPL and GPipe. Such an integration could potentially yield a hybrid approach that capitalizes on the benefits of both methodologies. By subdividing mini-batches and concurrently

Table 7: A comparison of the test accuracies of different methodologies when using different neural network architectures on CIFAR-10. We follow the same notations used in Table 4.

	Vanilla ConvNet	VGG	ResNet
BP	86.85 ± 0.57	93.02 ± 0.03	93.95 ± 0.11
Early Exit	83.16 ± 0.33	91.28 ± 0.15	89.63 ± 0.34
AL	$\mathbf{86.98} \pm 0.24$ †	$\mathbf{93.22} \pm 0.12$ †	91.33 ± 0.09
SCPL	$\mathbf{86.98} \pm 0.33$ †	$\mathbf{93.42} \pm 0.11$ †	$\mathbf{92.78} \pm 0.11$

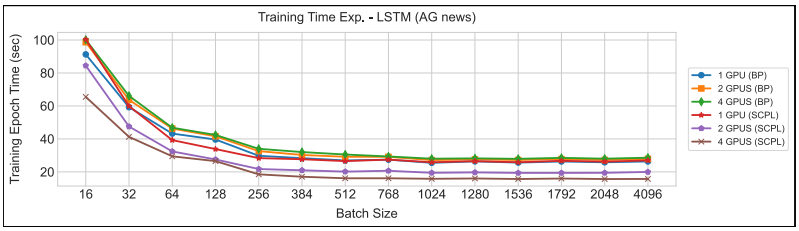


Figure 9: Empirical training time per epoch using LSTM architecture on AGNews.

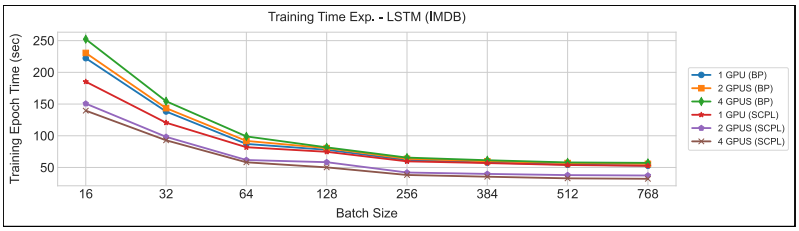


Figure 10: Empirical training time per epoch using LSTM architecture on IMDB.

designing local objectives, a harmonized pipeline structure may offer a solution to enhance training efficiency for large-scale neural network models.

The bottom subfigure of Figure 13 illustrates the integration of both SCPL and GPipe. Each mini-batch is divided into 3 micro-batches, so forward locking can be partially addressed, as demonstrated in t_1 to t_6. Additionally, since we allocate the local objective for each component using SCPL, each GPU can compute the local objective for each component and further compute the local gradients without waiting for the gradient information computed by other GPUs. In this example, the integration needs 22 time steps to complete one iteration of forward, backward, and parameter update, whereas SCPL and GPipe need 24 time steps and 31 time steps, respectively.

A.6 PSEUDO CODE

To help understand the details of SCPL, here are pseudocodes for local supervised contrastive losses (Algorithm 1) and SCPL without pipelining (Algorithm 2).

Table 8: A comparison of the test accuracies of different methodologies when using different neural network architectures on CIFAR-100. We follow the same notations used in Table 4.

	Vanilla ConvNet	VGG	ResNet
BP	58.68 ± 0.13	72.58 ± 0.39	73.59 ± 0.11
Early Exit	50.64 ± 0.44	71.11 ± 0.95	64.48 ± 0.41
AL	53.06 ± 0.15	72.43 ± 0.27	67.53 ± 0.32
SCPL	$\mathbf{59.63} \pm 0.37$ †	$\mathbf{73.14} \pm 0.30$ †	$\mathbf{70.41} \pm 0.27$

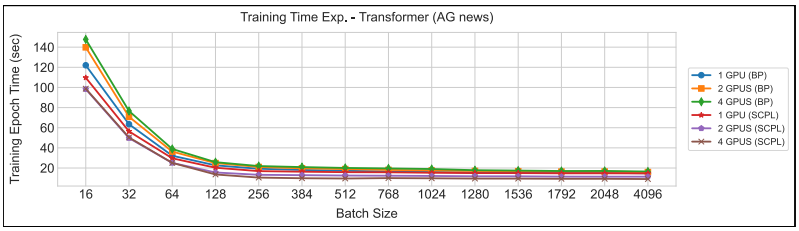


Figure 11: Empirical training time per epoch using Transformer architecture on AGNews.

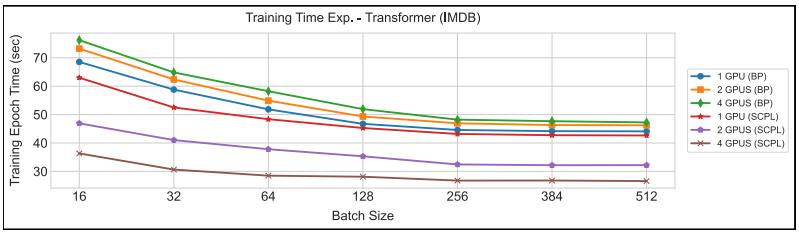


Figure 12: Empirical training time per epoch using Transformer architecture on IMDB.

```
import torch
import torch.nn as nn

class SupConLoss(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.linear = nn.Sequential(nn.Linear(dim, 512), nn.ReLU(), nn.
    Linear(512, 1024))
        self.temperature = 0.1

    def forward(self, x, label):
        x = self.linear(x)
        x = nn.functional.normalize(x)
        label = label.view(-1, 1)
        bsz = label.shape[0]
        mask = torch.eq(label, label.T).float()
        anchor_mask = torch.scatter(torch.ones_like(mask), 1, torch.
    arange(bsz).view(-1, 1), 0)
        logits = torch.div(torch.mm(x, x.T), self.temperature) deno =
    torch.exp(logits) * anchor_mask
        prob = logits - torch.log(deno.sum(1, keepdim=True))
        loss = -(anchor_mask * mask * prob).sum(1) / mask.sum()
        return loss.view(1, bsz).mean()
```

Algorithm 1: PyTorch-like pseudocode for L^{sc}

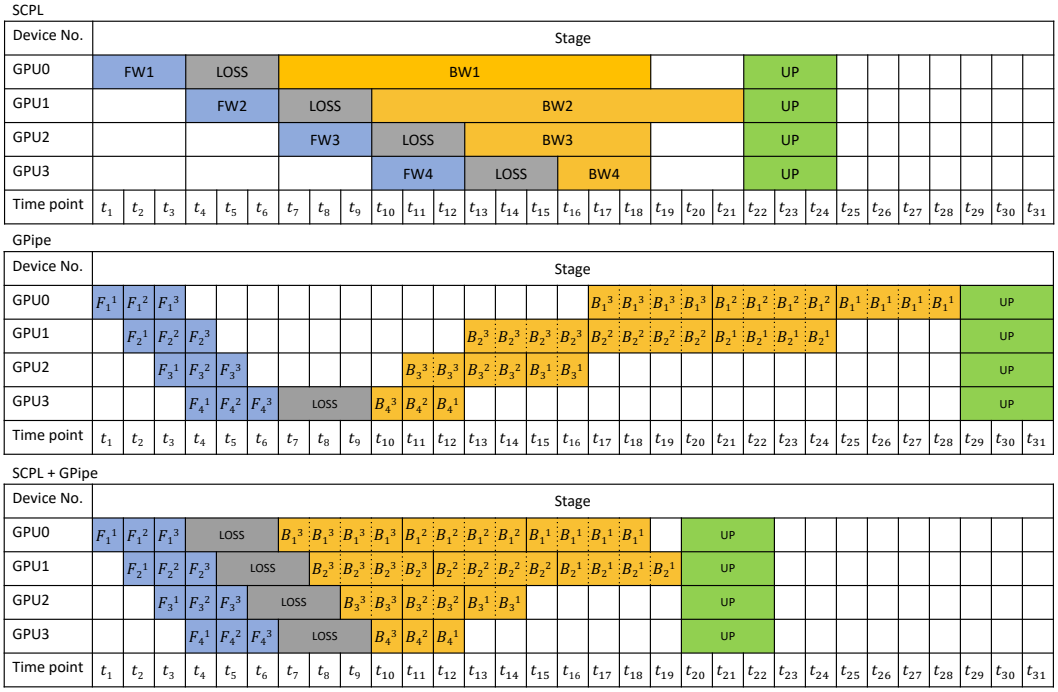


Figure 13: A comparison of SCPL, GPipe, and an integration of both.

```
import torch
import torch.nn as nn

# A simple 3-layer CNN example for SCPL architecture.
class CNN_SCPL(nn.Module):
    def __init__(self, dim):
        super().__init__()
        CNNs = [ ]
        losses = [ ]
        channels = [3, 128, 256, 512] self.shape = 32
        for i in range(3):
            CNNs.append(nn.Sequential(nn.Conv2d(channels[i], channels[i
    +1], padding=1), nn.ReLU())
            losses.append(SupConLoss(self.shape*self.shape*channels[i+1])
    )
        self.CNN = nn.ModuleList(CNNs)
        self.loss = nn.ModuleList(losses)
        self.fc = nn.Sequential(flatten(), nn.Linear(self.shape*self.
    shape*channels[-1], 10))
        self.ce = nn.CrossEntropyLoss()

    def forward(self, x, label): loss = 0
        for i in range(3):
            # .detach() prevents a gradient flows to neighboring layer
            x = self.CNN[i](x.detach())
            if self.training:
                loss += self.loss[i](x, label)
        y = self.fc(x.detach())
        if self.training:
            loss += self.ce(y, label)
            return loss
        return y
```

Algorithm 2: PyTorch-like pseudocode for SCPL without pipelining