Efficiency-and-the-Hardware-Software-Dichotomy

October 13, 2025

1 Efficiency / accuracy Pareto curves (accuracy vs FLOPs / latency)

We benchmarked AFMP against two widely used and well-optimized baselines: the convolutional ResNet family and the transformer-based DeiT family. The goal of this analysis is to highlight an often-overlooked distinction between theoretical computational cost and real-world performance. Specifically, we compared models both in terms of theoretical GFLOPs and measured inference latency.

The results are presented in last figure. This figure shows not only where AFMP fits in the broader landscape but also provides clear evidence of a fundamental mismatch between algorithmic efficiency and hardware execution. We refer to this as the "Latency–FLOPs Dichotomy,'' which represents the central finding of our empirical work. The analysis underscores that FLOPs alone cannot capture efficiency: AFMP's error-driven sparse routing creates a very different latency profile compared to dense attention, which directly motivated the design of this framework.

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
import numpy as np
from scipy.spatial import Delaunay
import time
import matplotlib.pyplot as plt
import os
import json
from PIL import Image

from tqdm import tqdm
```

[ ]:
```
# Author : zaryab Rahman
# Data: 25/9/2025 at university of malakand iqbal hostel
```

2 Static Graph Builder

[ ]:
```python
class StaticGraphBuilder:
    def __init__(self, mode='delaunay', window_size=8):
        self.mode = mode
        self.window_size = window_size
        self.cache = {}

    def build(self, positions):
        N = positions.shape[0]
        hash_key = f"{N}_{positions.mean().item():.4f}"

        if hash_key in self.cache:
            return self.cache[hash_key]

        if self.mode == 'delaunay' and positions.shape[1] > 1:
            tri = Delaunay(positions.cpu().numpy())
            edges = set()
            for simplex in tri.simplices:
                for i in range(3):
                    for j in range(i+1, 3):
                        a, b = sorted([simplex[i], simplex[j]])
                        edges.add((a, b))
            edge_index = torch.tensor(list(edges)).long().t().contiguous()

        elif self.mode == 'window' or positions.shape[1] == 1:
            edge_index = []
            for i in range(N):
                start = max(0, i - self.window_size)
                end = min(N, i + self.window_size + 1)
                for j in range(start, end):
                    if i != j:
                        edge_index.append([i, j])
            edge_index = torch.tensor(edge_index).long().t().contiguous()

        row, col = edge_index
        edge_attr = torch.norm(positions[row] - positions[col], dim=1)

        self.cache[hash_key] = (edge_index, edge_attr)
        return edge_index, edge_attr
```

3 Predictive Router

[ ]:
```
class PredictiveRouter(nn.Module):
    def __init__(self, d_model, k_dim=64, sparsity=0.3, is_sparse=True):
        super().__init__()
        self.d_model = d_model
        self.k_dim = k_dim
        self.sparsity = sparsity
        self.is_sparse = is_sparse
        self.W_pred = nn.Linear(d_model, d_model)
        self.W_q = nn.Linear(d_model, k_dim)
        self.W_k = nn.Linear(d_model, k_dim)
        self.U = nn.Parameter(torch.randn(k_dim, k_dim) * 0.02)
        self.v = nn.Sequential(
            nn.Linear(2 * d_model + 2, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )
        self.W_gate = nn.Linear(1, d_model)

    def forward(self, H, edge_index, edge_attr, batch_size,top_down=None):
        row, col = edge_index
        N, E = H.shape[0], edge_index.shape[1]

        if top_down is not None:
            pred_j = self.W_pred(top_down[row])
            errors = torch.norm(H[col] - pred_j, dim=1, keepdim=True)
        else:
            errors = torch.zeros(E, 1, device=H.device)

        Q = self.W_q(H[row])
        K = self.W_k(H[col])
        bilinear = (Q @ self.U) * K
        bilinear = bilinear.sum(dim=1, keepdim=True)



        feat_linear = torch.cat([
            H[row],
            H[col],
            errors,
            edge_attr.unsqueeze(1)
        ], dim=1)


        linear_term = self.v(feat_linear)
        scores = bilinear + linear_term
```

```
        num_nodes_per_graph = H.shape[0] // batch_size
        if self.is_sparse and self.sparsity > 0:
            k = int(self.sparsity * num_nodes_per_graph)
            if k > 0:
                node_scores = torch.full_like(scores, -float('inf'))
                num_edges_to_keep = k * batch_size
                if scores.numel() > num_edges_to_keep:
                    threshold = torch.topk(scores.view(-1), num_edges_to_keep).
↪values[-1]
                    mask = (scores >= threshold).float()
                    scores = scores * mask

        gates = torch.sigmoid(self.W_gate(scores))
        return gates
```

4 PCU

[ ]:
```
class PCU(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.W_self = nn.Linear(d_model, d_model)
        self.W_msg = nn.Linear(d_model, d_model)
        self.activation = nn.GELU()

    def forward(self, H, gates, edge_index):
        row, col = edge_index
        self_part = self.W_self(H)
        messages = self.W_msg(H[col])
        modulated = messages * gates

        agg = torch.zeros_like(H)
        agg = agg.index_add_(0, row, modulated)
        return self.activation(self_part + agg)
```

5 Top-Down Aggregator

[ ]:
```
class TopDownAggregator(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.W_t = nn.Linear(d_model, d_model)

    def forward(self, H_lower, H_higher, hierarchy):
        parent_signals = hierarchy @ H_higher
        modulation = self.W_t(parent_signals)
```

```
        return H_lower + modulation
```

6 Single layer

[ ]:
```
class AFMPLayer(nn.Module):
    def __init__(self, d_model, d_position, sparsity=0.3, is_sparse=True):
        super().__init__()
        self.router = PredictiveRouter(d_model, sparsity=sparsity,␣
↪is_sparse=is_sparse)
        self.pcu = PCU(d_model)

    def forward(self, H, edge_index, edge_attr, batch_size, top_down=None):
        batch_size, num_nodes, d_model = H.shape

        H_flat = H.view(-1, d_model) # Shape: (B*N, D)

        top_down_flat = None
        if top_down is not None:
            top_down_flat = top_down.view(-1, d_model)


        gates = self.router(H_flat, edge_index, edge_attr,␣
↪batch_size,top_down_flat)

        H_updated_flat = self.pcu(H_flat, gates, edge_index)

        H_out = H_updated_flat.view(batch_size, num_nodes, d_model)


        return H_out, gates
```

7 Regularizer

[ ]:
```
class SparsityRegularizer(nn.Module):
    def __init__(self, lambda_sparse=0.01):
        super().__init__()
        self.lambda_sparse = lambda_sparse

    def forward(self, gates):
        return self.lambda_sparse * torch.mean(torch.abs(gates))
```

8 Full model

[ ]:
```
class AFMP(nn.Module):
    def __init__(self, num_layers, d_model, d_position, num_classes,
                 img_size=32, patch_size=8, sparsity=0.3, is_sparse=True,
                 use_top_down=True,
                 lambda_sparse=1e-4):
        super().__init__()
        self.patch_size = patch_size
        self.img_size = img_size
        self.use_top_down = use_top_down

        self.num_patches = (img_size // patch_size) ** 2
        self.embed = nn.Linear(3 * patch_size * patch_size, d_model)
        self.position_embed = nn.Linear(2, d_model)

        grid = torch.arange(0, img_size, patch_size) + patch_size / 2
        grid_x, grid_y = torch.meshgrid(grid, grid, indexing='ij')
        static_positions = torch.stack([grid_x, grid_y], dim=-1).reshape(-1, 2)

        graph_builder = StaticGraphBuilder()
        edge_index, edge_attr = graph_builder.build(static_positions)

        self.register_buffer('static_positions', static_positions)
        self.register_buffer('edge_index', edge_index)
        self.register_buffer('edge_attr', edge_attr)

        self.layers = nn.ModuleList([
            AFMPLayer(d_model, d_position, sparsity, is_sparse)
            for _ in range(num_layers)
        ])

        if self.use_top_down:
            self.top_down_aggregators = nn.ModuleList([
                TopDownAggregator(d_model) for _ in range(num_layers-1)
            ])

        self.readout = nn.Sequential(
            nn.AdaptiveAvgPool1d(1),
            nn.Flatten(),
            nn.Linear(d_model, num_classes)
        )

        self.reg = SparsityRegularizer(
            lambda_sparse=lambda_sparse if is_sparse else 0.0
        )
```

```python
    def forward(self, images, return_internals=False):
        batch_size = images.size(0)
        p = self.patch_size

        patches = images.unfold(2, p, p).unfold(3, p, p)
        patches = patches.permute(0, 2, 3, 1, 4, 5)
        patches = patches.contiguous().view(batch_size, -1, 3 * p * p)

        pos_emb = self.position_embed(self.static_positions.expand(batch_size, -1, -1))
        H = self.embed(patches) + pos_emb

        # bottom up pass
        # the predictive routing still needs top-down signals during the bottom-up pass
        # this part remains the same to ensure a fair comparison of the final refinement step.
        states = []
        all_gates = []
        current = H
        for i, layer in enumerate(self.layers):
            # the top-down signal for routing comes from the PREVIOUS layer's output
            top_down_signal_for_routing = states[i-1] if i > 0 else None
            current, gates = layer(current, self.edge_index, self.edge_attr, batch_size, top_down=top_down_signal_for_routing)
            states.append(current)
            all_gates.append(gates)

        # Conditionally execute the Top-Down Refinement Pass
        if self.use_top_down:
            for i in range(len(self.layers)-2, -1, -1):
                hierarchy = self.create_hierarchy_map(states[i].shape[1], states[i+1].shape[1])
                hierarchy = hierarchy.to(images.device)
                # The states are refined in-place
                states[i] = self.top_down_aggregators[i](states[i], states[i+1], hierarchy)

        # readout from the final state (which is either refined or not)
        logits = self.readout(states[-1].permute(0, 2, 1)).squeeze(-1)

        sparsity_loss = self.reg(torch.cat(all_gates)) if all_gates else torch.tensor(0.0, device=images.device)

        if return_internals:
```

```
            return logits, sparsity_loss, all_gates, states
        else:
            return logits, sparsity_loss

    def create_hierarchy_map(self, n_low, n_high):

        if n_low == 0 or n_high == 0: return torch.zeros(n_low, n_high)
        if n_low % n_high != 0:
            ratio = n_low // n_high
        else:
            ratio = n_low // n_high
        if ratio == 0: return torch.zeros(n_low, n_high)

        map_matrix = torch.zeros(n_low, n_high)
        for i in range(n_high):
            start_idx = i * ratio
            end_idx = (i + 1) * ratio if i < n_high - 1 else n_low
            map_matrix[start_idx:end_idx, i] = 1.0
        return map_matrix
```

9 Different AFMP-Models from tiny to Base

[2]:
```
def get_afmp_family(num_classes=10,
                    img_size=224,
                    sparsity=0.3,
                    lambda_sparse=1e-4,
                    use_top_down=True,
                    is_sparse=True):
    """
    creates a dictionary containing different sizes of the AFMP model,
    configured with the optimal hyperparameters found in ablation studies.
    """

    configs = {
        'AFMP-XT': {'num_layers': 3, 'd_model': 96},   # extra Tiny
        'AFMP-T':  {'num_layers': 4, 'd_model': 128},  # tiny
        'AFMP-S':  {'num_layers': 5, 'd_model': 256},  # small
        'AFMP-B':  {'num_layers': 6, 'd_model': 384},  # base
    }

    afmp_family = {}
    print("--- Creating AFMP Model Family (with champion hyperparameters) ---")
    for name, model_config in configs.items():
        print(f"Building {name}...")

        model = AFMP(
```

```
            num_layers=model_config['num_layers'],
            d_model=model_config['d_model'],
            d_position=2,
            num_classes=num_classes,
            img_size=img_size,
            patch_size=16,
            sparsity=sparsity,
            lambda_sparse=lambda_sparse,
            is_sparse=is_sparse,
            use_top_down=use_top_down
        )
        afmp_family[name] = model

    return afmp_family
```

10 Baslines Models

[10]:
```
import timm
import torchvision.models as models

def get_baseline_families(num_classes=10, img_size=224):
    """
    creates a dictionary containing different sizes of baseline models (DeiT,␣
 ↪ResNet, ConvNeXt).
    """

    # deit family
    deit_configs = {
        'DeiT-T': 'deit_tiny_patch16_224',
        'DeiT-S': 'deit_small_patch16_224',

    }

    deit_family = {}
    print("\nCreating DeiT Model Family...")
    for name, model_name in deit_configs.items():
        print(f"Building {name}...")
        model = timm.create_model(
            model_name,
            pretrained=False,
            num_classes=num_classes,
            img_size=img_size
        )
        deit_family[name] = model

    # resnet family
```

```
    resnet_configs = {
        'ResNet-18': models.resnet18,
        'ResNet-34': models.resnet34,
        'ResNet-50': models.resnet50,
    }

    resnet_family = {}
    print("\nCreating ResNet Model Family...")
    for name, model_builder in resnet_configs.items():
        print(f"Building {name}...")
        model = model_builder(num_classes=num_classes)
        resnet_family[name] = model

    # avoid these models due to ther huge size but you can check it by urselves␣
↪if your computaion power allows thankyou
    '''
    # --- ConvNeXt Family (from timm) ---
    convnext_configs = {
        'ConvNeXt-T': 'convnext_tiny',
        'ConvNeXt-S': 'convnext_small',
        'ConvNeXt-B': 'convnext_base',
    }

    convnext_family = {}
    print("\ncreating ConvNeXt Model Family...")
    for name, model_name in convnext_configs.items():
        print(f"Building {name}...")
        model = timm.create_model(
            model_name,
            pretrained=False,
            num_classes=num_classes,
        )
        convnext_family[name] = model
    '''
    # combine all baseline families into one dictionary
    baseline_families = {**deit_family, **resnet_family,}

    return baseline_families
```

11 Calculate params

[4]:
```
def print_param_counts(model_dict):
    """
    prints a formatted table of model names and their trainable parameter␣
↪counts.
```

```python
    """
    print("\n" + "="*50)
    print(f"{'Model Name':<20} | {'Trainable Parameters (M)':<20}")
    print("-"*50)

    # sort models by parameter count for a cleaner table
    sorted_models = sorted(model_dict.items(), key=lambda item: sum(p.numel() 
 ↪for p in item[1].parameters() if p.requires_grad))

    for name, model in sorted_models:
        num_params = sum(p.numel() for p in model.parameters() if p.
 ↪requires_grad)
        print(f"{name:<20} | {num_params / 1e6:<20.2f}")

    print("="*50)


if __name__ == '__main__':
    NUM_CLASSES = 10
    IMG_SIZE = 224

    afmp_models = get_afmp_family(num_classes=NUM_CLASSES, img_size=IMG_SIZE)
    baseline_models = get_baseline_families(num_classes=NUM_CLASSES, 
 ↪img_size=IMG_SIZE)

    all_models = {**afmp_models, **baseline_models}

    print_param_counts(all_models)
```

```
--- Creating AFMP Model Family (with champion hyperparameters) ---
Building AFMP-XT…
Building AFMP-T…
Building AFMP-S…
Building AFMP-B…

--- Creating DeiT Model Family ---
Building DeiT-T…
Building DeiT-S…

--- Creating ResNet Model Family ---
Building ResNet-18…
Building ResNet-34…
Building ResNet-50…

--- Creating ConvNeXt Model Family ---
Building ConvNeXt-T…
Building ConvNeXt-S…
```

```
Building ConvNeXt-B…

==================================================
Model Name           | Trainable Parameters (M)
--------------------------------------------------
AFMP-XT              | 0.27
AFMP-T               | 0.50
AFMP-S               | 1.80
AFMP-B               | 4.32
DeiT-T               | 5.53
ResNet-18            | 11.18
ResNet-34            | 21.29
DeiT-S               | 21.67
ResNet-50            | 23.53
ConvNeXt-T           | 27.83
ConvNeXt-S           | 49.46
ConvNeXt-B           | 87.58
==================================================
```

[11]:

```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import time
import json
import os
from tqdm import tqdm
from fvcore.nn import FlopCountAnalysis

DRIVE_SAVE_PATH = r"C:\Users\labuo\OneDrive\Documents\AFMP_Paret"
RESULTS_FILE = os.path.join(DRIVE_SAVE_PATH, "pareto_results.json")
EPOCHS = 100 BATCH_SIZE = 256
LEARNING_RATE = 1e-3
NUM_CLASSES = 10
IMG_SIZE = 224

os.makedirs(DRIVE_SAVE_PATH, exist_ok=True)


# data preparation


def prepare_cifar10(batch_size, img_size):
    print("Preparing CIFAR-10 dataset...")
    transform_train = transforms.Compose([
        transforms.Resize((img_size, img_size)),
        transforms.TrivialAugmentWide(),
```

```
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
    transform_val = transforms.Compose([
        transforms.Resize((img_size, img_size)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    train_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
    val_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_val)

    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True)
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True)

    return train_loader, val_loader


# train and val loop

def train_epoch(model, loader, optimizer, device):
    model.train()
    total_loss = 0.0
    progress_bar = tqdm(loader, desc="[Train]", leave=False)
    for inputs, targets in progress_bar:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        if isinstance(outputs, tuple): outputs = outputs[0] # handle afmp output
        loss = nn.CrossEntropyLoss()(outputs, targets)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        progress_bar.set_postfix(loss=f"{loss.item():.4f}")
    return total_loss / len(loader)

def validate(model, loader, device):
    model.eval()
    total_correct, total_samples = 0, 0
    progress_bar = tqdm(loader, desc="[Val]", leave=False)
```

```
    with torch.no_grad():
        for inputs, targets in progress_bar:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            if isinstance(outputs, tuple): outputs = outputs[0]
            _, predicted = outputs.max(1)
            total_correct += predicted.eq(targets).sum().item()
            total_samples += inputs.size(0)
            progress_bar.set_postfix(acc=f"{100.*total_correct/total_samples:.
↪2f}%")
    return 100. * total_correct / total_samples




# measurement funtions



def measure_flops(model, device, img_size):
    model.eval()
    input_tensor = torch.randn(1, 3, img_size, img_size).to(device)
    try:
        flops = FlopCountAnalysis(model, input_tensor).total()
        return flops / 1e9  # Return GFLOPs
    except Exception as e:
        print(f"Could not calculate FLOPs: {e}")
        return -1.0

def measure_latency(model, device, img_size):
    model.eval()
    input_tensor = torch.randn(1, 3, img_size, img_size).to(device)
    with torch.no_grad():
        for _ in range(50): _ = model(input_tensor) # warm-up
        torch.cuda.synchronize()
        start_time = time.time()
        for _ in range(200): _ = model(input_tensor)
        torch.cuda.synchronize()
        end_time = time.time()
    return ((end_time - start_time) / 200) * 1000



# main wrapper
```

```python
def train_and_evaluate_model(model_name, model, train_loader, val_loader,␣
↪device, epochs, lr):
    print(f"\nTraining {model_name} for {epochs} epochs...")
    model.to(device)

    optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=5e-5)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,␣
↪T_max=epochs)

    best_acc = 0.0
    history = {'train_loss': [], 'val_acc': []}
    best_model_path = os.path.join(DRIVE_SAVE_PATH, f"{model_name}_best.pth")

    for epoch in range(epochs):
        train_loss = train_epoch(model, train_loader, optimizer, device)
        val_acc = validate(model, val_loader, device)
        scheduler.step()

        history['train_loss'].append(train_loss)
        history['val_acc'].append(val_acc)

        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Val␣
↪Acc: {val_acc:.2f}% | Best Acc: {max(best_acc, val_acc):.2f}%")

        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), best_model_path)
            print(f"  -> New best model saved to {best_model_path}")

    return best_acc, history


if __name__ == '__main__':
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    train_loader, val_loader = prepare_cifar10(BATCH_SIZE, IMG_SIZE)

    all_results = {}
    if os.path.exists(RESULTS_FILE):
        print(f"--- Loading existing results from {RESULTS_FILE} ---")
        with open(RESULTS_FILE, 'r') as f:
            all_results = json.load(f)

    afmp_models = get_afmp_family(num_classes=NUM_CLASSES, img_size=IMG_SIZE)
    baseline_models = get_baseline_families(num_classes=NUM_CLASSES,␣
↪img_size=IMG_SIZE)
    all_models = {**afmp_models, **baseline_models}
```

```python
    for model_name, model in all_models.items():
        if model_name in all_results:
            print(f"\n--- Results for {model_name} already exist. Skipping.␣
↪---")
            continue

        best_acc, history = train_and_evaluate_model(
            model_name, model, train_loader, val_loader, device, EPOCHS,␣
↪LEARNING_RATE
        )

        best_model_path = os.path.join(DRIVE_SAVE_PATH, f"{model_name}_best.
↪pth")
        if os.path.exists(best_model_path):
            model.load_state_dict(torch.load(best_model_path))

        print(f"--- Measuring FLOPs and Latency for {model_name} ---")
        gflops = measure_flops(model, device, IMG_SIZE)
        latency_ms = measure_latency(model, device, IMG_SIZE)
        params_m = sum(p.numel() for p in model.parameters() if p.
↪requires_grad) / 1e6

        print(f"Results for {model_name}: Acc={best_acc:.2f}%, FLOPs={gflops:.
↪2f}G, Latency={latency_ms:.2f}ms")

        # 4. Store all results
        all_results[model_name] = {
            'best_accuracy': best_acc,
            'gflops': gflops,
            'latency_ms': latency_ms,
            'params_m': params_m,
            'history': history,
        }

        print(f"--- Saving updated results to {RESULTS_FILE} ---")
        with open(RESULTS_FILE, 'w') as f:
            json.dump(all_results, f, indent=4)

    print("\n\n" + "="*50)
    print("ALL EXPERIMENTS COMPLETE!")
    print(f"Final results for all models saved in {RESULTS_FILE}")
    print("="*50)
```

```
Preparing CIFAR-10 dataset…
Files already downloaded and verified
Files already downloaded and verified
```

```
--- Loading existing results from
C:\Users\labuo\OneDrive\Documents\AFMP_Paret\pareto_results.json ---
--- Creating AFMP Model Family (with champion hyperparameters) ---
Building AFMP-XT…
Building AFMP-T…
Building AFMP-S…
Building AFMP-B…

--- Creating DeiT Model Family ---
Building DeiT-T…
Building DeiT-S…

--- Creating ResNet Model Family ---
Building ResNet-18…
Building ResNet-34…
Building ResNet-50…

--- Results for AFMP-XT already exist. Skipping. ---

--- Results for AFMP-T already exist. Skipping. ---

--- Results for AFMP-S already exist. Skipping. ---

--- Results for AFMP-B already exist. Skipping. ---

--- Results for DeiT-T already exist. Skipping. ---

--- Results for DeiT-S already exist. Skipping. ---

--- Results for ResNet-18 already exist. Skipping. ---

--- Results for ResNet-34 already exist. Skipping. ---

--- Results for ResNet-50 already exist. Skipping. ---


==================================================
ALL EXPERIMENTS COMPLETE!
Final results for all models saved in
C:\Users\labuo\OneDrive\Documents\AFMP_Paret\pareto_results.json
==================================================
```

[13]:

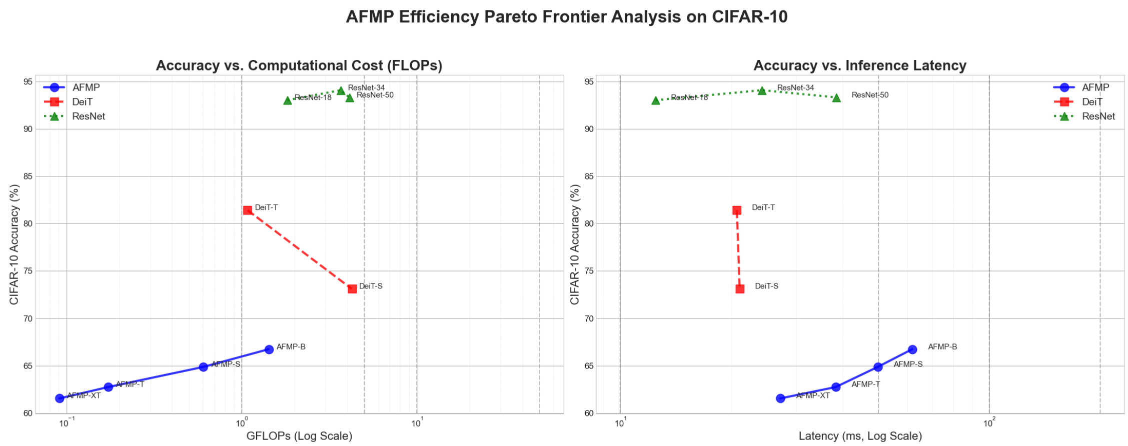
```python
import json
import matplotlib.pyplot as plt
import numpy as np
import os
```

```
RESULTS_FILE = r'C:\Users\labuo\OneDrive\Documents\AFMP_Paret\pareto_results.
↪json'

if not os.path.exists(RESULTS_FILE):
    print(f"ERROR: Results file not found at '{RESULTS_FILE}'")
    print("Please update the path and ensure the file exists.")
else:
    with open(RESULTS_FILE, 'r') as f:
        all_results = json.load(f)


    model_families = {
        'AFMP': {'color': 'blue', 'marker': 'o', 'linestyle': '-', 'models':␣
↪[]},
        'DeiT': {'color': 'red', 'marker': 's', 'linestyle': '--', 'models':␣
↪[]},
        'ResNet': {'color': 'green', 'marker': '^', 'linestyle': ':', 'models':␣
↪[]},
        # you can add ConvNeXt here if you run it later
        # 'ConvNeXt': {'color': 'purple', 'marker': 'D', 'linestyle': '-.',␣
↪'models': []},
    }

    print("--- Processing Results ---")
    for name, data in all_results.items():
        if 'best_accuracy' not in data or 'gflops' not in data:
            print(f"Skipping '{name}' because it lacks essential data (e.g.,␣
↪accuracy or FLOPs).")
            continue

        family_name = name.split('-')[0]
        if family_name in model_families:
            print(f"Found and processed data for: {name}")
            model_families[family_name]['models'].append({
                'name': name,
                'acc': data['best_accuracy'],
                'gflops': data['gflops'],
                'latency': data['latency_ms'],
                'params': data['params_m']
            })
        else:
            print(f"Warning: Could not determine family for model '{name}'.␣
↪Skipping.")

    for family in model_families.values():
```

```
        family['models'].sort(key=lambda x: x['gflops'])


    # olotting funtions
    def plot_pareto_curves(families):
        plt.style.use('seaborn-v0_8-whitegrid')
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(22, 9))

        ax1.set_title('Accuracy vs. Computational Cost (FLOPs)', fontsize=20,␣
↪weight='bold')
        ax1.set_xlabel('GFLOPs (Log Scale)', fontsize=16)
        ax1.set_ylabel('CIFAR-10 Accuracy (%)', fontsize=16)
        ax1.set_xscale('log')
        ax1.grid(which='major', linestyle='-', linewidth='0.5', color='gray')
        ax1.grid(which='minor', linestyle=':', linewidth='0.5',␣
↪color='lightgray')

        for gflop_marker in [1, 5, 10, 50]:
            ax1.axvline(x=gflop_marker, color='gray', linestyle='--', alpha=0.5)

        ax2.set_title('Accuracy vs. Inference Latency', fontsize=20,␣
↪weight='bold')
        ax2.set_xlabel('Latency (ms, Log Scale)', fontsize=16)
        ax2.set_ylabel('CIFAR-10 Accuracy (%)', fontsize=16)
        ax2.set_xscale('log')
        ax2.grid(which='major', linestyle='-', linewidth='0.5', color='gray')
        ax2.grid(which='minor', linestyle=':', linewidth='0.5',␣
↪color='lightgray')

        for lat_marker in [10, 50, 100, 200]:
            ax2.axvline(x=lat_marker, color='gray', linestyle='--', alpha=0.5)

        for family_name, details in families.items():
            if not details['models']: continue

            names = [m['name'] for m in details['models']]
            accs = [m['acc'] for m in details['models']]
            gflops = [m['gflops'] for m in details['models']]
            latencies = [m['latency'] for m in details['models']]

            ax1.plot(gflops, accs, marker=details['marker'],␣
↪linestyle=details['linestyle'],
                     color=details['color'], label=family_name, markersize=12,␣
↪lw=3, alpha=0.8)
            ax2.plot(latencies, accs, marker=details['marker'],␣
↪linestyle=details['linestyle'],
```

```
                    color=details['color'], label=family_name, markersize=12,␣
↪lw=3, alpha=0.8)

            for i, name in enumerate(names):
                ax1.text(gflops[i] * 1.1, accs[i], name, fontsize=11, ha='left')
                ax2.text(latencies[i] * 1.1, accs[i], name, fontsize=11,␣
↪ha='left')

        ax1.legend(fontsize=14)
        ax2.legend(fontsize=14)
        ax1.tick_params(axis='both', which='major', labelsize=12)
        ax2.tick_params(axis='both', which='major', labelsize=12)

        fig.suptitle('AFMP Efficiency Pareto Frontier Analysis on CIFAR-10',␣
↪fontsize=24, weight='bold')
        plt.tight_layout(rect=[0, 0.03, 1, 0.94])

        save_path = os.path.join(os.path.dirname(RESULTS_FILE), 'pareto_curves.
↪png')
        plt.savefig(save_path, dpi=300)
        print(f"\nPareto curve plots saved to '{save_path}'")
        plt.show()

    plot_pareto_curves(model_families)
```

```
--- Processing Results ---
Found and processed data for: AFMP-XT
Found and processed data for: AFMP-T
Found and processed data for: AFMP-S
Found and processed data for: AFMP-B
Found and processed data for: DeiT-T
Found and processed data for: DeiT-S
Found and processed data for: ResNet-18
Found and processed data for: ResNet-34
Found and processed data for: ResNet-50

Pareto curve plots saved to
'C:\Users\labuo\OneDrive\Documents\AFMP_Paret\pareto_curves.png'
```



[15]:
```
!nvidia-smi
```

```
Mon Aug 18 15:17:57 2025
+-----------------------------------------------------------------------------------------
----------+
| NVIDIA-SMI 566.07                 Driver Version: 566.07         CUDA Version:
12.7     |
|-----------------------------------------+------------------------+------------
----------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util
Compute M. |
|                                         |                        |
MIG M. |
|=========================================+========================+============
==========|
|   0  NVIDIA GeForce RTX 3070 ...  WDDM  |   00000000:01:00.0 Off |
N/A |
| N/A   42C    P8             11W /   95W |    7975MiB /   8192MiB |      0%
Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------+------------
----------+

+-----------------------------------------------------------------------------------------
----------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name
GPU Memory |
```

```
|        ID   ID
Usage      |
|===============================================================================
==========|
|    0   N/A  N/A     16020      C   …buo\miniconda3\envs\AFMP\python.exe
N/A      |
+-------------------------------------------------------------------------------
----------+
```

[ ]:
```
!pip install fvcore
```

```
Collecting fvcore
  Downloading fvcore-0.1.5.post20221221.tar.gz (50 kB)
                           0.0/50.2

kB ? eta -:--:--
                      50.2/50.2 kB 4.9

MB/s eta 0:00:00
  Preparing metadata (setup.py) … done
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(from fvcore) (2.0.2)
Collecting yacs>=0.1.6 (from fvcore)
  Downloading yacs-0.1.8-py3-none-any.whl.metadata (639 bytes)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-
packages (from fvcore) (6.0.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages
(from fvcore) (4.67.1)
Requirement already satisfied: termcolor>=1.1 in /usr/local/lib/python3.11/dist-
packages (from fvcore) (3.1.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.11/dist-packages
(from fvcore) (11.3.0)
Requirement already satisfied: tabulate in /usr/local/lib/python3.11/dist-
packages (from fvcore) (0.9.0)
Collecting iopath>=0.1.7 (from fvcore)
  Downloading iopath-0.1.10.tar.gz (42 kB)
                           42.2/42.2 kB
4.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) … done
Requirement already satisfied: typing_extensions in
/usr/local/lib/python3.11/dist-packages (from iopath>=0.1.7->fvcore) (4.14.1)
Collecting portalocker (from iopath>=0.1.7->fvcore)
  Downloading portalocker-3.2.0-py3-none-any.whl.metadata (8.7 kB)
Downloading yacs-0.1.8-py3-none-any.whl (14 kB)
Downloading portalocker-3.2.0-py3-none-any.whl (22 kB)
Building wheels for collected packages: fvcore, iopath
  Building wheel for fvcore (setup.py) … done
  Created wheel for fvcore: filename=fvcore-0.1.5.post20221221-py3-none-any.whl
size=61397
```

```
sha256=10d26a4fb8c17e9189dd07e620920f76d94e80c8619f7a5c1c894ab4db5a5d7c
  Stored in directory: /root/.cache/pip/wheels/65/71/95/3b8fde5c65c6e4a806e0867c
1651dcc71a1cb2f3430e8f355f
  Building wheel for iopath (setup.py) … done
  Created wheel for iopath: filename=iopath-0.1.10-py3-none-any.whl size=31527
sha256=c357913e77a2575487f5ab99733ee05822ef821865bb011b4fa8dd8242551f06
  Stored in directory: /root/.cache/pip/wheels/ba/5e/16/6117f8fe7e9c0c161a795e10
d94645ebcf301ccbd01f66d8ec
Successfully built fvcore iopath
Installing collected packages: yacs, portalocker, iopath, fvcore
Successfully installed fvcore-0.1.5.post20221221 iopath-0.1.10 portalocker-3.2.0
yacs-0.1.8
```