Complete Reproduction Guide: Token Reduction Performance Analysis

Tuesday, October 10 – October 20, 2025

Institute of Electronics
National Yang Ming Chiao Tung University

Presenter: Hao-Chun, Liang
Supervisor: Bo-Cheng Lai, Ph.D

NATIONAL
YANG MING CHIAO TUNG
UNIVERSITY

Environment Setting

1. Clone the Repository

```
# Clone from your repository
mkdir -p ~/project && cd ~/project
git clone https://github.com/noyaboy/TokenReductionPT.git
cd TokenReductionPT

# Check you're on the main branch with all analysis commits
git log --oneline | head -10
```

You should see commits like:

```
8b0099e Update gitignore to exclude intermediate profiling files
109d75b Add NCU profiling analysis and findings
2d63c18 Add comprehensive profiling analysis and final report
d144192 HYPOTHESIS VALIDATED: TR performance depends on input size
...
```

Environment Setting

2. Set Up Python Environment

```
# Create conda environment (recommended)
conda create -n tr_analysis python=3.9
conda activate tr_analysis

# Or use venv
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt
pip install matplotlib
pip install pynvml

cp -r /home/noah/project/TokenReductionPT/configs/datasets/ ./configs/
cp -r /home/noah/data ~/.

# Verify PyTorch CUDA support
python -c "import torch; print(f'CUDA available: {torch.cuda.is_available()}')"
python -c "import torch; print(f'CUDA version: {torch.version.cuda}')"
```

Environment Setting

3. Verify Environment

Run a quick sanity check:

```
# Quick test run (should complete in ~30 seconds)
python speed_test.py \
    --cfg configs/cub_ft_weakaugs.yaml \
    --model deit_base_patch16_224.fb_in1k \
    --input-size 224 \
    --debugging \
    --batch-size 1 \
    --test_multiple 1 \
    --warmup_iters 10 \
    --dummy_loader
```

Expected output format:

```
run_name,tp,latency_ms,time_total,flops,max_memory,no_params,no_params_trainable,power
cub_...,150.5,6.64,0.11,17.57,1234.5,85.8,85.8,150.2
```

Initial Observations

- **Configuration 1: Low-Resolution, Small Batch**

```
# Baseline (no TR)
python speed_test.py --model deit_base_patch16_224.fb_in1k \
    --input-size 224 --batch-size 1 ...
# Result: ~170 img/s, 17.57 GFLOPs

# With TR (keep_rate=0.1)
python speed_test.py --model topk_deit_base_patch16_224.fb_in1k \
    --input-size 224 --batch-size 1 --keep_rate 0.1 --reduction_loc 3 6 9 ...
# Result: ~141 img/s, 5.55 GFLOPs
# Conclusion: 68% FLOPs reduction but 17% SLOWER ❌
```

Initial Observations

- **Configuration 2: High-Resolution, Large Batch**
  - Paradox: Similar FLOPs Reduction, Completely Opposite Performance Outcomes!

```
# Baseline (no TR)
python speed_test.py --model deit_base_patch16_224.fb_in1k \
    --input-size 448 --batch-size 8 ...
# Result: ~109 img/s, 78.52 GFLOPs

# With TR (keep_rate=0.1)
python speed_test.py --model topk_deit_base_patch16_224.fb_in1k \
    --input-size 448 --batch-size 8 --keep_rate 0.1 --reduction_loc 3 6 9 ...
# Result: ~321 img/s, 24.73 GFLOPs
# Conclusion: 69% FLOPs reduction and 3x FASTER ✅
```

Initial Hypotheses

- **Batch Size Hypothesis**
  - Small Batches Can't Saturate GPU → Overhead Dominates
    - **Prediction: TR Should Improve With Larger Batches**
  - Small Workloads Have Poor SM Occupancy
    - **Prediction: Performance Should Correlate With GPU Utilization Metrics**
- **Input Size Hypothesis**
  - Token Count Determines Savings vs Overhead Trade-Off
    - **Prediction: TR Should Improve With Larger Inputs Regardless of Batch Size**
  - TR Adds Many Small Operations
    - **Prediction: Kernel Count Should Correlate With Launch Overhead**

Phase 1: Hypothesis Formation and Testing

- **Objective**
  - Test Batch Size or Input Sizels Primary Determinant of Performance.
- **Experiment Design**
  - Hypothesis: Input Size (Token Count) Is Primary Factor, Not Batch Size.
- **Test**
  - Run Comprehensive Batch Size Sweeps at Two Different Input Sizes:
    - Input Size = 224: Batch Sizes = 1, 2, 4, 8, 16
    - Input Size = 448: Batch Sizes = 1, 2, 4, 8
- **Prediction**
  - TR Behavior Consistent Within Each Input Size, Regardless of Batch Size

Step 1: Batch Size Sweep at Input Size = 224

- Test Baseline (No TR) at Batch Size = 1, 2, 4, 8, 16
- Test With TR (Keep Rate = 0.1) at Batch Size = 1, 2, 4, 8, 16
- Save Results to Batch_Size_Sweep_Results.csv
- Display Comparison Table

Navigate to the profiling directory:

```
cd profiling_results
```

Run the is=224 sweep:

```
bash 07_test_intermediate_batches.sh
```

Expected output:

```
Comparison (Speedup with TR):
================================================
Batch Size | Baseline TP | TR TP | Speedup | Better?
------------------------------------------------
         1 |      152.70 | 137.30 |   0.899x | ✗ NO
         2 |      157.90 | 136.40 |   0.864x | ✗ NO
         4 |      149.80 | 132.00 |   0.881x | ✗ NO
         8 |      157.70 | 136.70 |   0.867x | ✗ NO
        16 |      157.40 | 132.80 |   0.844x | ✗ NO
================================================
```

Key observation: TR is consistently 10-16% slower across ALL batch sizes at Input Size = 224.

Step 2: Batch Size Sweep at Input Size = 448

- Modify the Existing One …
- **Key Observations:**
  - Token Reduction Is 13.9x Faster at Batch Size =1 (Most Dramatic Speedup!)
  - Token Reduction Provides 2–3x Speedup for Most Batch Sizes
  - Anomaly: Batch Size = 4 Is the Only Case Where Token Reduction Is Slower

Expected output:

```
Comparison (Speedup with TR):
================================================
Batch Size | Baseline TP | TR TP | Speedup | Better?
------------------------------------------------
         1 |        7.60 | 105.30 |  13.868x | ✓ YES
         2 |       32.80 |  79.30 |   2.418x | ✓ YES
         4 |       71.40 |  61.60 |   0.863x | ✗ NO (anomaly)
         8 |       32.80 |  73.60 |   2.244x | ✓ YES
================================================
```

Step 3: Hypothesis Validation

- **Result: ✅Hypothesis Validated**
  - Input Resolution Is the Primary Factor, Not Batch Size!

Input Size	Batch Size	Baseline TP	TR TP	Speedup	Latency Δ	Baseline GFLOPs	TR GFLOPs	Tokens (pre→post)
224	1	152.7	137.3	0.90x	+11.3%	17.57	5.55	196 → 20
224	2	157.9	136.4	0.86x	+15.8%	17.57	5.55	196 → 20
224	4	149.8	132.0	0.88x	+13.3%	17.57	5.55	196 → 20
224	8	157.7	136.7	0.87x	+15.5%	17.57	5.55	196 → 20
224	16	157.4	132.8	0.84x	+18.5%	17.57	5.55	196 → 20
448	**1**	**7.6**	**105.3**	**13.87x**	**-92.8%**	**78.52**	**24.73**	**784 → 78**
448	2	32.8	79.3	2.42x	-58.6%	78.52	24.73	784 → 78
448	4	71.4	61.6	0.86x	+15.8%	78.52	24.73	784 → 78
448	8	32.8	73.6	2.24x	-55.4%	78.52	24.73	784 → 78

Phase 2: Timeline Profiling (NSYS)

- **Objective**
  - Use NVIDIA NSYS to Profile GPU Execution Timelines and Quantify:
    - **Total GPU Kernel Execution Time**
    - **Number of Kernel Launches**
    - **Kernel Launch Overhead**

Step 1: Quick Profiling Setup

- **This Profiles:**
  - Input Size 224, Batch Size 1, Baseline – Fast Baseline
  - Input Size 224, Batch Size 1, With TR – Slow Despite FLOPs Reduction
  - Input Size 448, Batch Size 8, Baseline – Slow Baseline
  - Input Size 448, Batch Size 8, With TR – Fast With TR

```
cd profiling_results

# Run quick profiling (4 configurations)
bash profile_quick_comparison.sh
```

Output files:

```
nsys_traces/quick_224_bs1_baseline.nsys-rep
nsys_traces/quick_224_bs1_with_tr.nsys-rep
nsys_traces/quick_448_bs8_baseline.nsys-rep
nsys_traces/quick_448_bs8_with_tr.nsys-rep
```

Step 2/3: SQLite Conversion and Analyze Profiling Data

Step 2: Convert to SQLite for Analysis

```
# Convert .nsys-rep files to .sqlite for programmatic analysis
cd nsys_traces

for file in quick_*.nsys-rep; do
    echo "Exporting $file to SQLite..."
    nsys export --type sqlite $file
done

# This creates .sqlite files alongside .nsys-rep files
ls -lh *.sqlite
```

Step 3: Analyze Profiling Data

Use the provided analysis script:

```
cd ..  # Back to profiling_results directory

python nsys_overhead_analysis.py > NSYS_ANALYSIS.txt
```

Step 3: Analyze Profiling Data

Configuration	GPU Time (ms)	Kernel Count	Avg Kernel (µs)	Real Latency (ms)
is=224 bs=1 baseline	154.34	6,856	22.51	6.55
is=224 bs=1 WITH TR	82.39	7,422	11.10	7.29
GPU speedup	**-46.6%**	**+8.3%**	**-51%**	**+11.3% (slower)**
is=448 bs=8 baseline	568.49	6,855	82.93	30.50
is=448 bs=8 WITH TR	243.60	7,370	33.05	13.59
GPU speedup	**-57.1%**	**+7.5%**	**-60%**	**-55.4% (faster)**

Step 4: Key Insights from NSYS

- **Critical Discovery: GPU vs System Performance Mismatch**
  - At Input Size 224, Batch Size 1:
    - **GPU Computation 46.6% Faster With TR (154.34 → 82.39 ms)**
    - **System Throughput 11.3% Slower With TR (6.55 → 7.29 ms/img)**
    - **Overhead >> Savings: ~72.7ms System Overhead Cancels 71.95 ms GPU Savings**
  - At Input Size 448, Batch Size 8:
    - **GPU Computation Is 57.1% Faster With TR (568.49 → 243.60 ms)**
    - **System Throughput Is 55.4% Faster With TR (30.50 → 13.59 ms/img)**
    - **Savings >> Overhead: GPU Savings of 324.89 ms Far Exceeds System Overhead**
  - Overhead Sources:
    - **+566 Additional Kernel Launches at Input Size 224 (~5–10 µs Each)**
    - **CPU-GPU Synchronization**
    - **Memory Allocation/Deallocation**
    - **Python/PyTorch Dispatching Overhead**
    - **Reduced GPU Occupancy From Smaller Kernels (22.51 → 11.10 µs Average)**

Phase 3: Kernel-Level Profiling (NCU)

- **Objective**
  - **Use NVIDIA NCU to Profile Individual GPU Kernels in Detail:**
    - SM (Streaming Multiprocessor) Utilization
    - Memory Bandwidth Utilization
    - Cache Hit Rates
    - Occupancy Levels
    - Compute vs Memory Boundedness

Step 1: Lightweight NCU Profiling

```
cd profiling_results

# Create lightweight NCU profiling script
cat > ncu_quick_profile.sh << 'EOF'
#!/bin/bash
mkdir -p ncu_reports

echo "NCU Profiling: is=224 bs=1 baseline"
ncu --set default --export ncu_reports/ncu_224_bs1_baseline \
    --force-overwrite \
    python ../speed_test.py \
        --cfg ../configs/cub_ft_weakaugs.yaml \
        --model deit_base_patch16_224.fb_in1k \
        --input-size 224 \
        --batch-size 1 \
        --debugging \
        --test_multiple 1 \
        --warmup_iters 20 \
        --dummy_loader
```

```
echo "NCU Profiling: is=224 bs=1 with TR"
ncu --set default --export ncu_reports/ncu_224_bs1_with_tr \
    --force-overwrite \
    python ../speed_test.py \
        --cfg ../configs/cub_ft_weakaugs.yaml \
        --model topk_deit_base_patch16_224.fb_in1k \
        --input-size 224 \
        --batch-size 1 \
        --keep_rate 0.1 \
        --reduction_loc 3 6 9 \
        --debugging \
        --test_multiple 1 \
        --warmup_iters 20 \
        --dummy_loader

echo "NCU Profiling complete!"
EOF

chmod +x ncu_quick_profile.sh
bash ncu_quick_profile.sh
```

Step 2: Analyze NCU Reports

```
# Extract key metrics from NCU reports
python << 'EOF'
import subprocess
import re

reports = [
    ('ncu_reports/ncu_224_bs1_baseline.ncu-rep', 'is=224 bs=1 baseline'),
    ('ncu_reports/ncu_224_bs1_with_tr.ncu-rep', 'is=224 bs=1 WITH TR'),
]

print("=" * 80)
print("NCU PROFILING ANALYSIS")
print("=" * 80)
print()

for report_path, name in reports:
    print(f"--- {name} ---")

    # Extract SM efficiency
    result = subprocess.run(
        ['ncu', '--import', report_path, '--csv', '--page', 'raw'],
        capture_output=True, text=True
    )

    # Parse metrics (simplified - actual parsing more complex)
    print(f"  Report: {report_path}")
    print(f"  Use NCU GUI for detailed analysis: ncu-ui {report_path}")
    print()

print("=" * 80)
EOF
```

Step 3: Key Metrics from NCU

Configuration	SM Utilization	Memory Utilization	Achieved Occupancy
is=224 bs=1 baseline	**3.20%**	28.85%	43.32%
is=224 bs=1 WITH TR	**3.24%**	29.09%	43.44%
is=448 bs=8 WITH TR	**4.34%**	39.00%	54.84%
is=448 bs=4 WITH TR (ANOMALY)	**4.40%**	39.13%	54.90%
is=448 bs=1 WITH TR (BEST)	**4.28%**	38.44%	55.19%

Configuration	L1/TEX Hit Rate	L2 Hit Rate
is=224 bs=1 baseline	**1.85%**	52.27%
is=224 bs=1 WITH TR	**1.85%**	51.89%
is=448 bs=8 WITH TR	**1.85%**	50.67%
is=448 bs=4 WITH TR (ANOMALY)	**1.85%**	50.33%
is=448 bs=1 WITH TR (BEST)	**1.85%**	50.51%

Step 3: Key Metrics from NCU

- **GPU Utilization (All Configurations)**
  - SM Utilization: 3–4% (Extremely Low!)
  - Memory Utilization: 29–39% (Not Saturated)
  - Achieved Occupancy: 43–55% (Moderate)
- **Interpretation**
  - Workload Is Launch-Bound, Not Compute-Bound
  - GPU Spends Most Time Idle Between Kernel Launches
  - Low Utilization Explains Why Adding Kernels Hurts Performance
- **Cache Performance**
  - L1 Hit Rate: ~2% (Terrible – Streaming Access Patterns)
  - L2 Hit Rate: ~50% (Moderate)
- **Interpretation**
  - Poor Data Locality
  - Mostly Streaming Reads Without Reuse
  - Memory Access Patterns Are Scattered

Phase 4: Analysis and Interpretation

- **Comprehensive Data From Three Sources:**
  - Batch Size Sweeps
    - **Revealed Input Size Dependency**
  - NSYS Profiling
    - **Quantified Kernel Launch Overhead and GPU Time**
  - NCU Profiling
    - **Showed Extremely Low GPU Utilization**

Conclusion

- **Main Findings**
  - Input Size Is Primary Factor
    - **Token Count (Not Batch Size) Determines TR Performance**
    - **Break-Even Point: 400 Tokens (320×320 Input)**
    - **Savings Scale Quadratically ($O(n^2)$) With Attention, Overhead Stays Fixed**
  - GPU–System Performance Mismatch
    - **GPU Computation Is Faster With TR at Both Input Sizes**
    - **System Overhead Cancels Out GPU Savings at Small Scales**
    - **Overhead Sources: +500 Kernel Launches, CPU–GPU Sync, Memory Management**
  - Extremely Low GPU Utilization
    - **Only 3–4% SM Utilization Across All Configs**
    - **Workload Is Launch-Bound, Not Compute-Bound**
    - **Explains Why Kernel Count Matters More Than Kernel Efficiency**
  - FLOPs Are Not Predictive
    - **Similar FLOPs Reduction (~68%) Yields Opposite Results**
    - **Real Performance Depends on System-Level Factors**
    - **Must Consider Overhead, Not Just Computational Complexity**
  - Batch Size Has Minimal Impact
    - **Only 5–6% Performance Variance Across 16× Batch Range at Fixed Input Size**
    - **Counterintuitive: Smallest Batch (Batch Size 1) Shows Largest Speedup at Input Size 448**
    - **TR Overhead Appears to Be Per-Sample, Not Per-Batch**

Recommendations

- **Use Token Reduction For:**
  - High-Resolution Images (≥448×448, ~784+ Tokens)
  - Dense Prediction Tasks (Segmentation, Detection)
  - Fine-Grained Classification Requiring Large Inputs
  - Video Frame Processing (Typically 720p+)
  - Medical Imaging (Often >1024×1024)
- **Avoid Token Reduction For:**
  - Standard ImageNet Classification (224×224, 196 Tokens)
  - Low-Resolution Tasks (≤256×256)
  - Maximum Accuracy Requirements (TR Costs 3–5% Accuracy)
  - Batch Size 4 at Input Size 448 (Specific Anomaly)