CSC 591/791 - Real Time AI Course Project

LPRNet Optimization Techniques

Team Members	**Unity ID**
Srivathsan Govindarajan	sgovind7
Vengatesh Deen Dayal	vdeenda

Report for Efficient Vehicle Plate Recognition

Link to GitHub Repo: Repo Link

Overview of the Optimization Project

This report details the optimization of the LPRNet deep learning model, which is specifically designed for real-time vehicle plate recognition. The primary objectives of the optimization efforts were to improve inference speed, reduce model size, and maintain high accuracy to facilitate deployment in resource-constrained environments.

Original DNN Model Architecture and Analysis

Model Description

The baseline model, **LPRNet**, is a lightweight convolutional neural network architecture tailored for the task of vehicle plate recognition. Designed to balance computational efficiency and predictive accuracy, LPRNet is particularly well-suited for mobile and embedded systems, where resources are limited.

1. Backbone Network: Initial convolutional layer (3 channels → 64 filters) Multiple small basic blocks for feature extraction Batch normalization layers for training stability Strategic placement of MaxPool3D layers for spatial reduction
2. Small Basic Blocks:
    a. Specialized convolutional blocks with the following sequence:
        i. 1x1 convolution for channel reduction
        ii. 3x1 followed by 1x3 convolutions for efficient feature extraction
        iii. Final 1x1 convolution for channel expansion
    b. ReLU activation functions between layers
3. Global Context Module:
    a. Features multiple parallel paths for context aggregation
    b. Implements average pooling at different scales

Baseline Performance Metrics:

1. Model Size: 1.705 MB
2. Test Accuracy: 89.9%
3. Average Inference Time: 0.264 seconds per sample Measured across 1000 test samples Running on Google Colab's CPU environment

Applied Optimizations

Model-Level Optimizations

1. Post Training Static Quantization

- **Technique**: Post-training quantization represents a sophisticated approach to model compression through numerical precision reduction. Our implementation utilized PyTorch's quantization framework with specific optimizations for x86 hardware. Converts both weight and activation values to INT8 precision.
- **Implementation**: The implementation follows 3 stages:
    - **Preparation Phase**:
        - Configuration of quantization parameters
        - Definition of observation points within the model
        - Setup of calibration methodology
    - **Calibration Process**:
        - Utilized 100 samples from the test dataset
        - Collected activation statistics for optimal scaling
        - Determined quantization parameters for each layer
    - **Model Conversion:**
        - Finalizes the quantization by replacing floating-point operations with their quantized equivalents (e.g., INT8 operations). This step applies the learned scale and zero-point values from calibration.
- **Objective**: Weights and activations are quantized to INT8, reducing memory. Because of this, quantized models typically execute faster on CPUs, especially for architectures supporting INT8 instructions. The calibration process ensures that the quantized model's accuracy remains close to the original floating-point model.
- **Results and Analysis:**
    - Size Reduction: 91.8% (from 1.705 MB to 0.139 MB)
    - Accuracy: 4.8% reduction (89.9% to 85.1%)
    - Speed Improvement: 89% reduction in inference time

2. 1:4 Structured Sparsity Pruning

- **Technique**: Applied structured pruning to enforce a 1:4 sparsity pattern, where one weight out of every four in a group was retained, and the others were zeroed out.
- **Implementation**: Custom pruning masks were generated and applied during training to adhere to the 1:4 sparsity structure. This technique took advantage of hardware acceleration capabilities optimized for such sparsity patterns.
- **Objective**: Enhance inference speed and reduce memory usage, particularly on hardware platforms optimized for structured sparsity.
- **Results and Analysis:**
    - Sparsity Achievement: 25% overall reduction due to structured sparsity
    - Size Impact: Reduced to 1.316 MB

    - Accuracy Trade-off: Significant drop to 68.6% due to the removal of weights at each layer
    - Inference time: 0.244 seconds

3. Channel Based Pruning

- **Technique**: Channel-based pruning targets the reduction of less significant channels in convolutional layers, thereby decreasing the model's computational load.
  Implementation: The pruning process involves computing the L1 norm of the weights associated with each channel. A sparsity parameter determines the percentage of channels to prune. The pruning mask is generated by comparing the channel norms against a threshold value calculated from the desired sparsity. The pruning operation was applied iteratively across convolutional layers using a custom `ChannelPruner` class to dynamically update the masks and apply them to the model's weights.
- **Objective**: Channel pruning significantly reduces the number of computations required for convolutional operations, thereby optimizing the model's runtime performance and memory usage.
- **Performance Analysis**:
    - Size Reduction: 45.7% (to 0.926 MB)
    - Accuracy Maintenance: 82.3%
    - Inference Time: 0.192 seconds

Although the inference time decreased due to pruning at each convolution layer, except the last one, we faced a significant drop in the test accuracy of the model

4. Weight Based Quantization

- **Technique**: Quantizes the model weights from float-based to INT8 precision
- **Implementation**: Quantization was implemented using a custom `WeightQuantizer` class, which calculated the scale factor and zero-point for converting weights to lower precision. Tensor values were clipped and rounded to fit within the specified 8-bit range. The method ensures efficient forward passes while maintaining the representational fidelity of the weights. Calibration of the model is not required.
- **Objective**: The primary objective of this quantization process was to minimize memory and computational requirements, particularly benefiting deployment on resource-constrained devices.
- **Performance Analysis**:
    - Size Reduction: Reduced to 0.25MB
    - Accuracy Maintenance: 90%
    - Inference Time: 0.04 seconds from 0.24s

MLC (Machine Learning Compilation) Optimizations

1. Auto-Tuning with TVM Relay

- **Technique**: TVM Relay facilitates both auto-tuning of operations and end-to-end graph compilation. Auto-tuning optimizes individual operator configurations, while graph compilation fuses operators and optimizes memory access patterns for efficient execution
- **Implementation**: The model, first converted into a TorchScript static graph, was transformed into Relay IR. Using TVM's auto-tuner, with XGBoost as the search strategy, operator-level configurations were optimized based on trial-and-error searches. Once optimal configurations were identified and recorded in autotuning.json, the Relay graph underwent further graph-level optimizations. This included kernel fusion, loop unrolling, and scheduling improvements to minimize latency and maximize throughput. The resulting graph was compiled into an optimized library for the target device (LLVM), tailored for its architecture.
    - **Configuration**:
        - Target: LLVM backend.
        - Optimization Level: 3.
        - Number of Trials: 20.
        - Early Stopping: Enabled after 100 iterations.
    - Tuning records were stored in `autotuning.json`.

  The optimization pipeline included:

    1. Graph Analysis Phase
        - Computational graph extraction
        - Operator pattern recognition
    2. Auto Tuning Process
        - Search space definition
        - Performance measurement
        - Configuration optimization
    3. Implementation step
- **Outcome**: The tuning process led to significant improvements in GFLOPS (billion floating-point operations per second), resulting in enhanced throughput.
- **Results:**
    - Accuracy Maintenance: 90%
    - Speed Improvement: 0.037 seconds inference time
    - Runtime Optimization: Significant reduction in computational overhead
    - Model Size : 1.705 MB

2. TensorRT Integration

- **Technique**: TensorRT optimizes GPU inference by combining multiple operations, such as convolutions and activations, into a single computational kernel (fusion), reducing overhead. Additionally, it applies reduced precision arithmetic (FP16 and INT8) to save memory and increase computational throughput while maintaining acceptable accuracy. TensorRT further optimizes the specific GPU architecture by selecting the most efficient kernel implementations and tuning memory layouts to leverage parallel processing and hardware acceleration capabilities.

- **Implementation**: The model was compiled using TensorRT, which performed graph optimization, including operator fusion and precision adjustments. TensorRT's inference engine replaced redundant computation with GPU-efficient paths. Specific configurations, such as kernel selection and memory layout optimizations, were automatically chosen by TensorRT based on GPU architecture.
- **Objective**: To maximize inference speed and GPU utilization while preserving accuracy. It specifically targeted GPU-bound tasks that benefit from TensorRT's ability to harness the GPU's parallelism.
- **Results:**
    - Accuracy Maintenance: 90%
    - Speed Improvement: 0.025 seconds inference time
    - Model Size : 1.705 MB

3. TorchScript JIT Optimization

- **Technique**: This optimization utilizes TorchScript to transform the model into a static computation graph. By freezing dynamic aspects of the computation graph, operations can be fused and execution overhead is minimized.
- **Implementation**: TorchScript traced the LPRNet model using sample input data to create an optimized graph representation. This representation enabled operation fusion, removing unnecessary computations and streamlining inference. As a result, the static graph was more efficient compared to the original, dynamic PyTorch computation.
- **Objective**: Reduce runtime latency and computational redundancy during inference.
- **Effects**:
    - Size Reduction: No significant change in model size.
    - Speed Improvement: Improved execution speed due to reduced dynamic overhead to 0.24s.
    - Accuracy Impact: Accuracy was unaffected, retaining the baseline performance of 90.00%.

4. Using torch.compile Optimization - AOT Compilation

- **Technique**: The torch.compile(model) functionality can be considered an Ahead-of-Time (AOT) Compilation Optimization. This is because it transforms the dynamic computation graph of a PyTorch model into a static, optimized graph that is precompiled for efficient execution.
- **Implementation**: The function torch.compile analyzed the LPRNet computation graph and applied graph-level transformations. This included operator fusion, kernel selection, and memory optimization. These adjustments ensured minimal runtime overhead and improved parallelism, dynamically adapting to the underlying hardware.

- **Objective**: Improve overall inference performance by reducing latency, optimizing memory usage, and streamlining graph execution.
- **Effects**:
    - Size Reduction: to 1.12 MB.
    - Speed Improvement: Improved execution speed to 0.09s.
    - Accuracy Impact: Accuracy was unaffected, retaining the baseline performance of 90.00%.

Performance Metrics Comparison

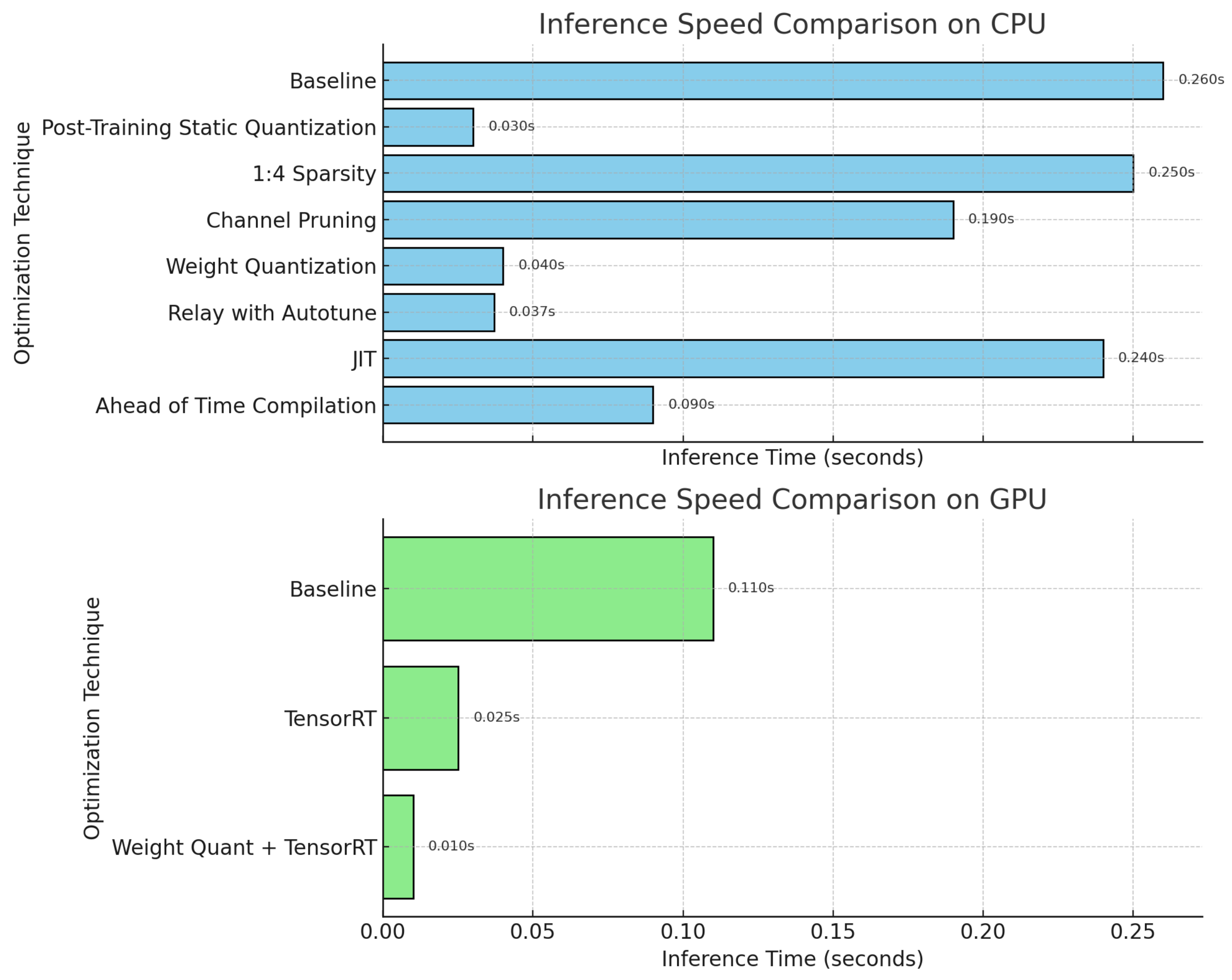
Optimization Technique	**Size(MB)**	**Accuracy(%)**	**Speed**	**Hardware Requirements**
Baseline	1.705 MB	90%	0.26s	CPU
			0.11s	GPU
Post-Training Static Quantization	0.13 MB	85 %	0.03s	CPU
1:4 Sparsity	1.315 MB	68%	0.25s	CPU
Channel Pruning	0.92 MB	82%	0.19s	CPU
Weight Quantization	0.25 MB	90%	0.04s	CPU
Relay with Autotune	1.705 MB	90%	0.037s	CPU
TorchScript JIT Optimization	-	90%	0.24s	CPU
TensorRT	1.705 MB	90%	0.025s	GPU
Ahead of Time Compilation	1.12 MB	89%	0.09s	CPU

Weight Quant + TensorRT	**0.25 MB**	**90%**	**0.01s**	**GPU**

Based on comprehensive analysis, we recommend a two-stage optimization pipeline:

1. Applying Weight Quantization
2. Optimizing the weight quantized model using TensorRT

This combination provided a significant drop in running time while matching baseline accuracy of 90%. The speed significantly reduced to 0.01s and the model size had a significant drop to 0.25MB.



Lessons Learned

1. **Proficiency in Optimization Frameworks:** Leveraging advanced tools like Relay and TensorRT requires a deep understanding of both hardware and software optimizations. The effort is worthwhile given the significant performance improvements achievable.

2. **Quantization Challenges:** While quantization effectively reduces computational demands and memory usage, careful calibration and validation are critical to ensuring model accuracy, particularly for sensitive applications.
3. **Pruning Tradeoffs:** Both channel pruning and 1:4 structured sparsity pruning proved highly effective in reducing computational complexity and memory requirements. These techniques failed to preserve accuracy while optimizing performance because of the removal of important weights in the DNN network. Pruning techniques should be applied with careful consideration while using it as the sole optimization technique.
4. **Hardware-Specific Optimizations:** TensorRT's tailored optimizations for NVIDIA GPUs underscore the value of leveraging platform-specific tools to achieve peak performance.

5. **Benefits of Auto-Tuning:** TVM's auto-tuning framework highlighted the power of automated task-level optimization, yielding substantial improvements in throughput and overall efficiency. For each operation, the auto-tuner evaluates various configurations to determine the most efficient execution settings. This involves testing parameters such as memory usage, loop tiling, and kernel placement to maximize hardware utilization. The optimization process is guided by a predictive algorithm, often using tree-based models like XGBoost, which intelligently prioritizes the most promising configurations.

Testing Instructions: Provided in the Github repo's ReadMe file

---

Conclusion

This report demonstrates the successful application of model-level and MLC optimizations to LPRNet, achieving substantial improvements in efficiency while preserving high accuracy. These results underscore the practicality of deploying compact DNNs for vehicle plate recognition in real-world settings.

References

1. https://github.com/sirius-ai/LPRNet_Pytorch/tree/master
2. https://pytorch.org/docs/stable/index.html
3. https://tvm.apache.org/docs/
4. https://docs.nvidia.com/deeplearning/tensorrt/