ORACLE

Improving Inference Performance of ML

with the Divide-and-Conquer Principle

Alex Kogan
Oracle Labs

ML models are everywhere ...

Image Recognition
for x-ray labeling

Speech Recognition
for voice search

Predictive Analytics
for drug development

Video Processing
for traffic monitoring

Text Generation
for chatbot interaction

… and they are often deployed on CPUs

Deep Learning inferencing at scale with Oracle Cloud A1 Compute with Elotl Luna

October 5, 2022 | 13 minute read

Kailas Jawadekar
Director of Product Marketing

Oracle Cloud Infrastructure Blog

Accelerating Stable Diffusion Inference on Intel CPUs

Published March 28, 2023

Update on GitHub

juliensimon Julien Simon
echarlaix Ella Charlaix

Hugging Face

How We Scaled Bert To Serve 1+ Billion Daily Requests on CPUs

Quoc N. Le · Follow
12 min read · May 27, 2020

ROBLOX

Optimizing BERT model for Intel CPU Cores using ONNX runtime default execution provider

March 1, 2021 • 5 min read

Microsoft Open Source Blog

ML models scale poorly when deployed on CPUs. Why?

On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not “enough” work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework “tax”
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms

ML models scale poorly when deployed on CPUs. Why?

On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not “enough” work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework “tax”
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms

ML models scale poorly when deployed on CPUs. Why?

On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not “enough” work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework “tax”
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms

ML models scale poorly when deployed on CPUs. Why?

On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not “enough” work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework “tax”
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms

ML models scale poorly when deployed on CPUs. Why?

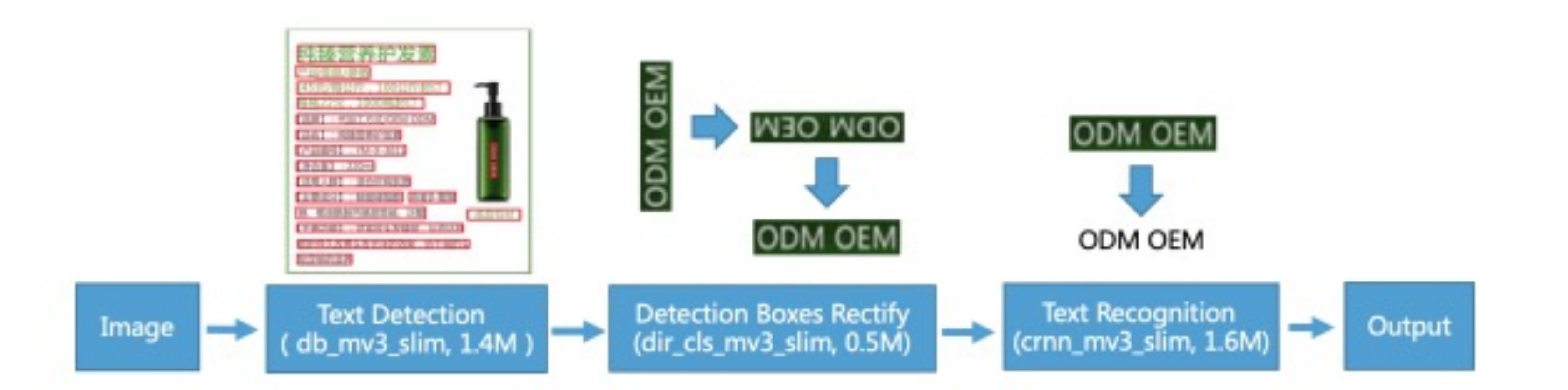
On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not "enough" work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework "tax"
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms

Pipeline model architecture

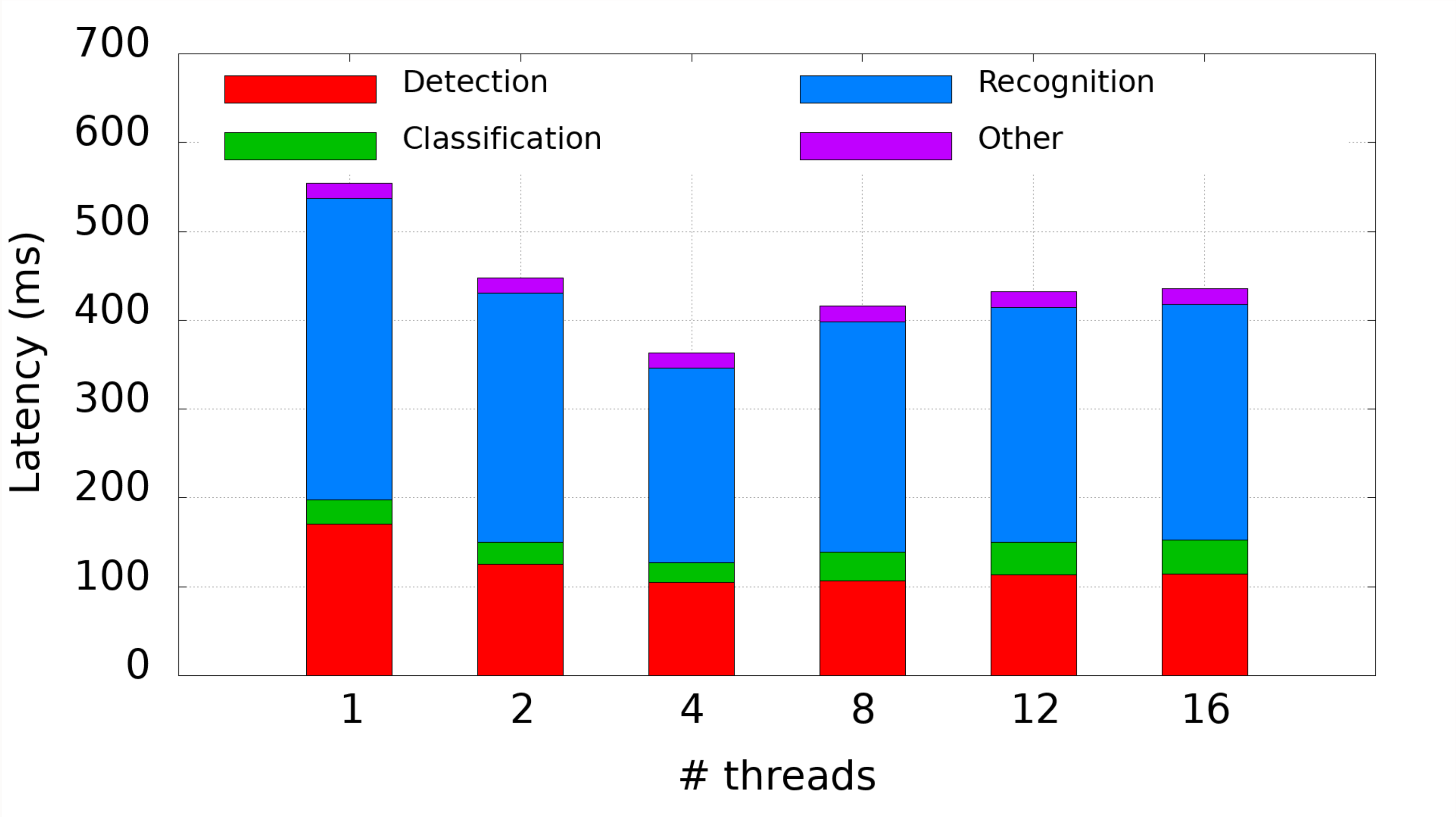
- Popular in image/video processing domains

PaddleOCR



* From Du et. al., “PP-OCR: A practical ultra lightweight OCR system”. CoRR, abs/2009.09941, 2020.

PaddleOCR performance



ML models scale poorly when deployed on CPUs. Why?

On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not “enough” work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework “tax”
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms

ML models scale poorly when deployed on CPUs. Why?

On a high-level: **ML frameworks are geared towards high performance training**, less so inference

- Not “enough” work
  - trained with large batches of large inputs, deployed with small batches of small inputs
- Non-Scalable operators
  - some have inherently bottlenecks, others are plain implementation bugs
- Framework “tax”
  - small overhead per every op adds up …
- Model architecture
  - when one phase of a pipeline does not scale, the entire pipeline underperforms
- Wasted work due to padding
  - batching is a double-edged sword

What can be done?

- Rewrite ML models
  - requires domain-specific expertise, retraining (time, cost)
  - no guarantee that performance will improve

or

- Optimize and tune ML framework
  - no changes to the model implementation
  - requires extensive profiling and engineering effort

or

- Break the problem into smaller pieces of work, and run them in parallel
  - simple idea that works well
  - requires minimal code changes

Divide-and-Conquer Principle

Divide-and-Conquer Principle (DACP) design

Given a computation job $J = \{j_1, j_2, \ldots, j_k\}$

- s.t. each independent part j_i can be executed in parallel with other parts

Assume we have an oracle assigning relative weight $w_i \in [0,1]$ to j_i

- e.g., corresponding to the number of FLOPS
- or single-thread latency

Assume we have C cores

Assign $c_i = \max \{ 1, \lfloor w_i * C \rfloor \}$ to j_i

- allocate c_i worker threads (=cores) for j_i

DACP design

Assign $c_i = \max \{ 1, \lfloor w_i * C \rfloor \}$ to j_i

- allocate c_i worker threads (=cores) for j_i

What if $\sum c_i > C$?

- might happen if k (number of job parts) > C
- not an issue – some jobs will run after others

What if $\sum c_i < C$?

- might happen due to $\lfloor \rfloor$
- sort all job parts by their remaining unallocated wight: $w_i * C - \lfloor w_i * C \rfloor$
- assign one core to each part, in the descending order, until all cores are allocated

DACP design

Given a computation job $J = \{j_1, j_2, \ldots, j_k\}$

- s.t. each independent part j_i can be executed in parallel with other parts

Assume we have an oracle assigning relative weight $w_i \in [0,1]$ to j_i

- e.g., corresponding to the number of FLOPS
- or single-thread latency

Assume we have C cores

Assign $c_i = \max \{ 1, \lfloor w_i * C \rfloor \}$ to j_i

- allocate c_i worker threads (=cores) for j_i

Sort all job parts by their remaining unallocated wight: $w_i * C - \lfloor w_i * C \rfloor$

Assign one core to each part, in the descending order, until all cores are allocated

How to implement the “weight assignment” oracle?

Idea 1: Employ profiling and lightweight classification

- run profiling during the warm-up phase and tune up the weights
- associate job parts of similar shapes/features to the weight obtained during profiling

Idea 2: Set the weight proportional to input tensor sizes

- let s_i be the size of input tensors for j_i
- set $w_i = s_i / \sum s_i$
- (simplistically) assume linear correlation between FLOPS and input tensor size
- no profiling is required

Implementing DACP in OnnxRT

- Extend InferenceSession API with prun
  - similar to run, but accepts a list of inputs and returns a list of outputs

- Internally, prun implements the DACP design
  - create one worker thread per input
    - and run those threads in parallel
  - each worker thread creates a thread pool …
    - set the size of the pool according to w_i
  - … and invokes the session's run method with that pool
- ~200 lines of code

User code changes

```
class TextRecognizer(object):
    def __init__(self, args):
        ...
        self.predictor = ort.InferenceSession(args.file_path)
        self.postprocess_op = build_post_process(args)
        ...

    def __call__(self, img_list):
        img_num = len(img_list)
        for beg_img_no in range(0, img_num, batch_num):
            end_img_no = min(img_num, beg_img_no + batch_num)

            inputs = prepare(img_list, beg_img_no, end_img_no)

            outputs = self.predictor.run(inputs)

            preds = outputs[0]
            rec_result = self.postprocess_op(preds)
            all_results.add(rec_result)

        return all_results
```

Listing 2. Original (shortened and edited for clarity) `TextRecognizer` class implementation from PaddleOCR

User code changes

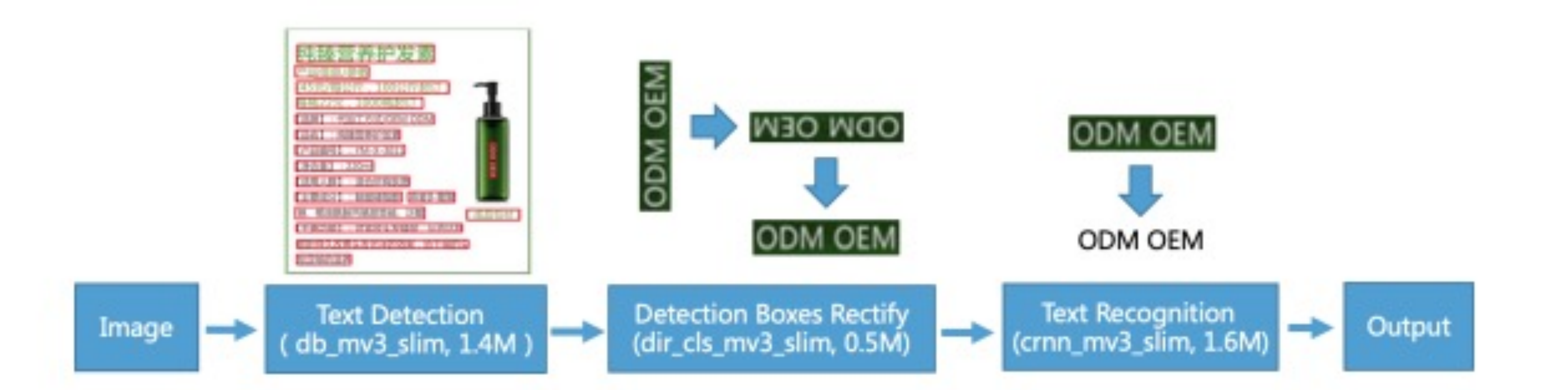
```
class TextRecognizer(object):
    def __init__(self, args):
        ...
        self.predictor = ort.InferenceSession(args.file_path)
        self.postprocess_op = build_post_process(args)
        ...

    def __call__(self, img_list):
        img_num = len(img_list)
        for beg_img_no in range(0, img_num, batch_num):
            end_img_no = min(img_num, beg_img_no + batch_num)

            inputs = prepare(img_list, beg_img_no, end_img_no)

            outputs = self.predictor.run(inputs)

            preds = outputs[0]
            rec_result = self.postprocess_op(preds)
            all_results.add(rec_result)

        return all_results
```

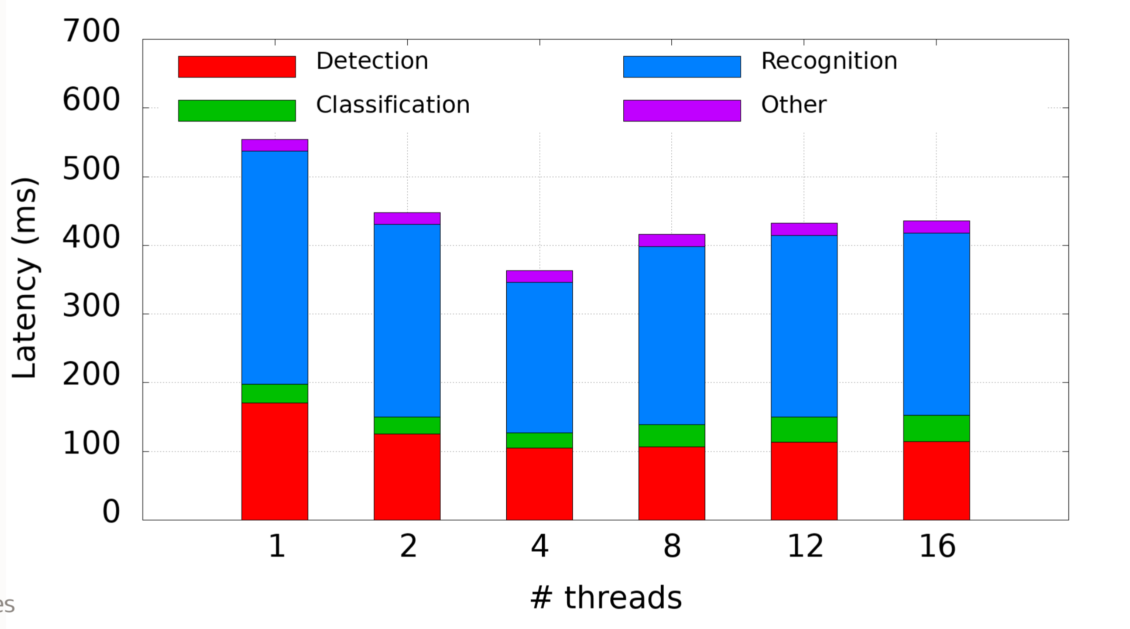
Listing 2. Original (shortened and edited for clarity) `TextRecognizer` class implementation from PaddleOCR

```
class TextRecognizer(object):
    def __init__(self, args):
        ...
        self.predictor = ort.InferenceSession(args.file_path)
        self.postprocess_op = build_post_process(args)
        ...

    def __call__(self, img_list):
        img_num = len(img_list)
        for beg_img_no in range(0, img_num, batch_num):
            end_img_no = min(img_num, beg_img_no + batch_num)

            inputs = prepare(img_list, beg_img_no, end_img_no)
            all_inputs.append(inputs)
        all_outputs = self.predictor.prun(all_inputs)
        for outputs in all_outputs:
            preds = outputs[0]
            rec_result = self.postprocess_op(preds)
            all_results.add(rec_result)

        return all_results
```

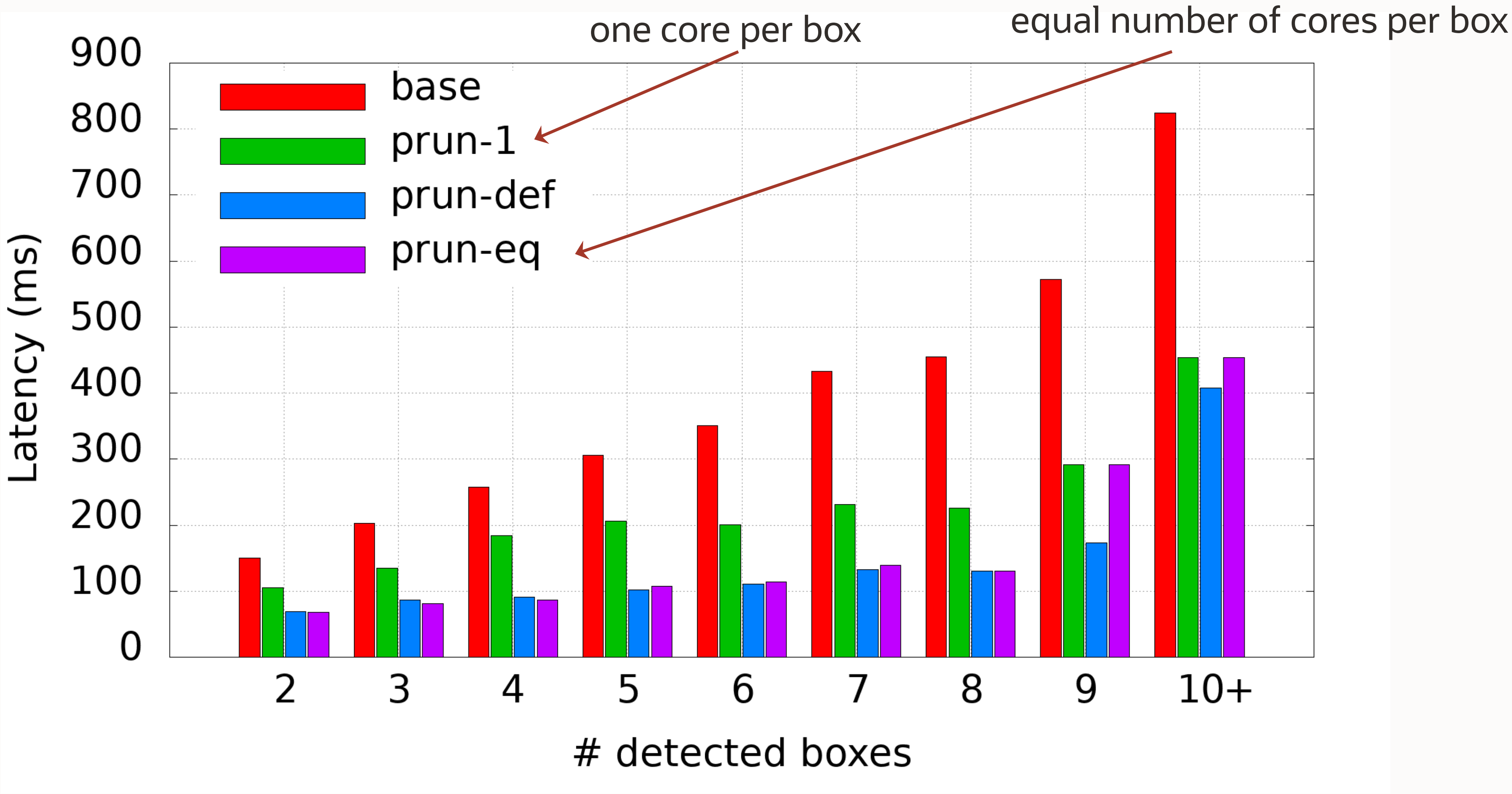
Listing 3. Modified `TextRecognizer` class implementation (uses prun). Added or modified lines are in red

DACP evaluation: PaddleOCR (recap)

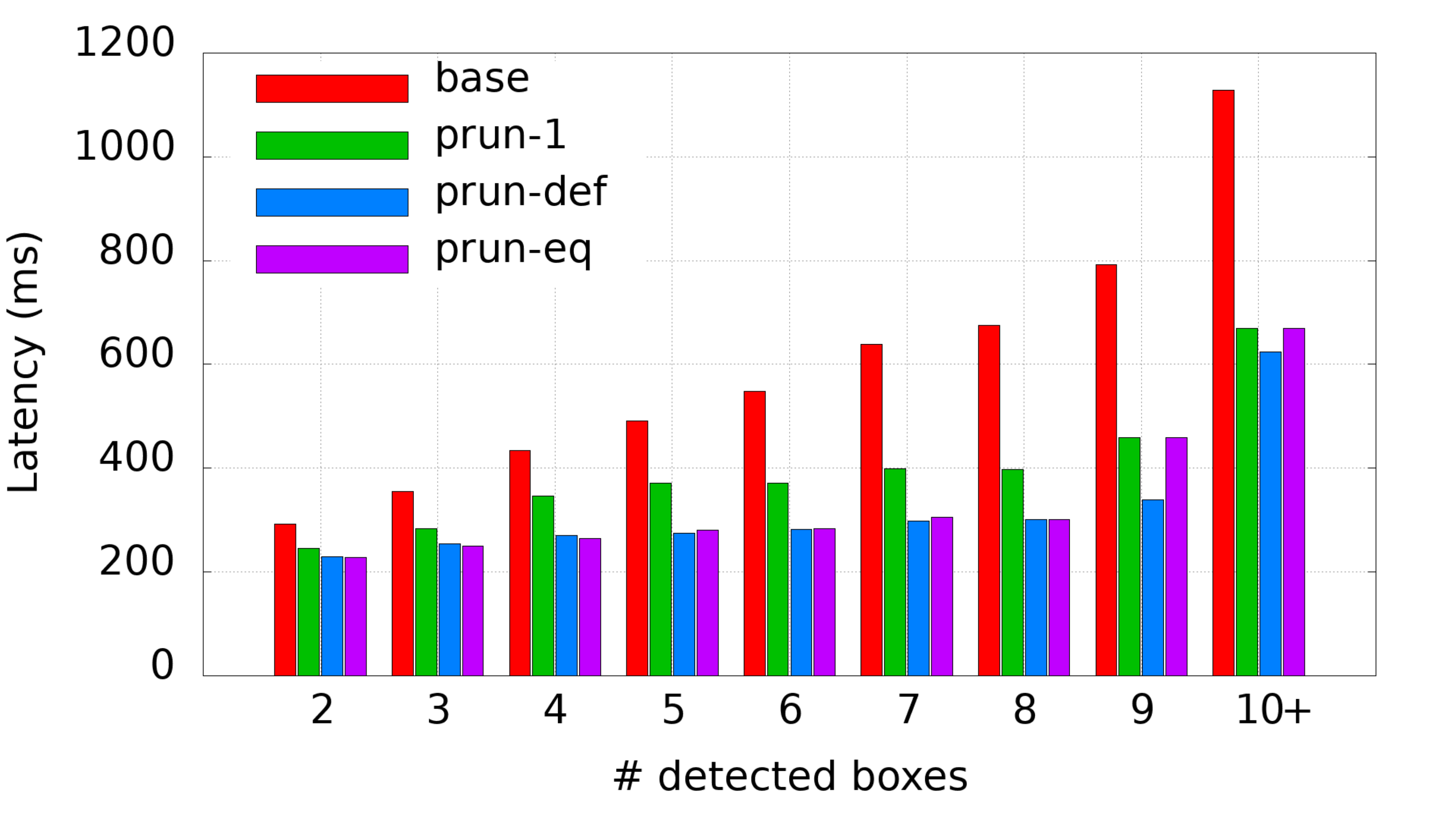




DACP evaluation: PaddleOCR / Text Recognition

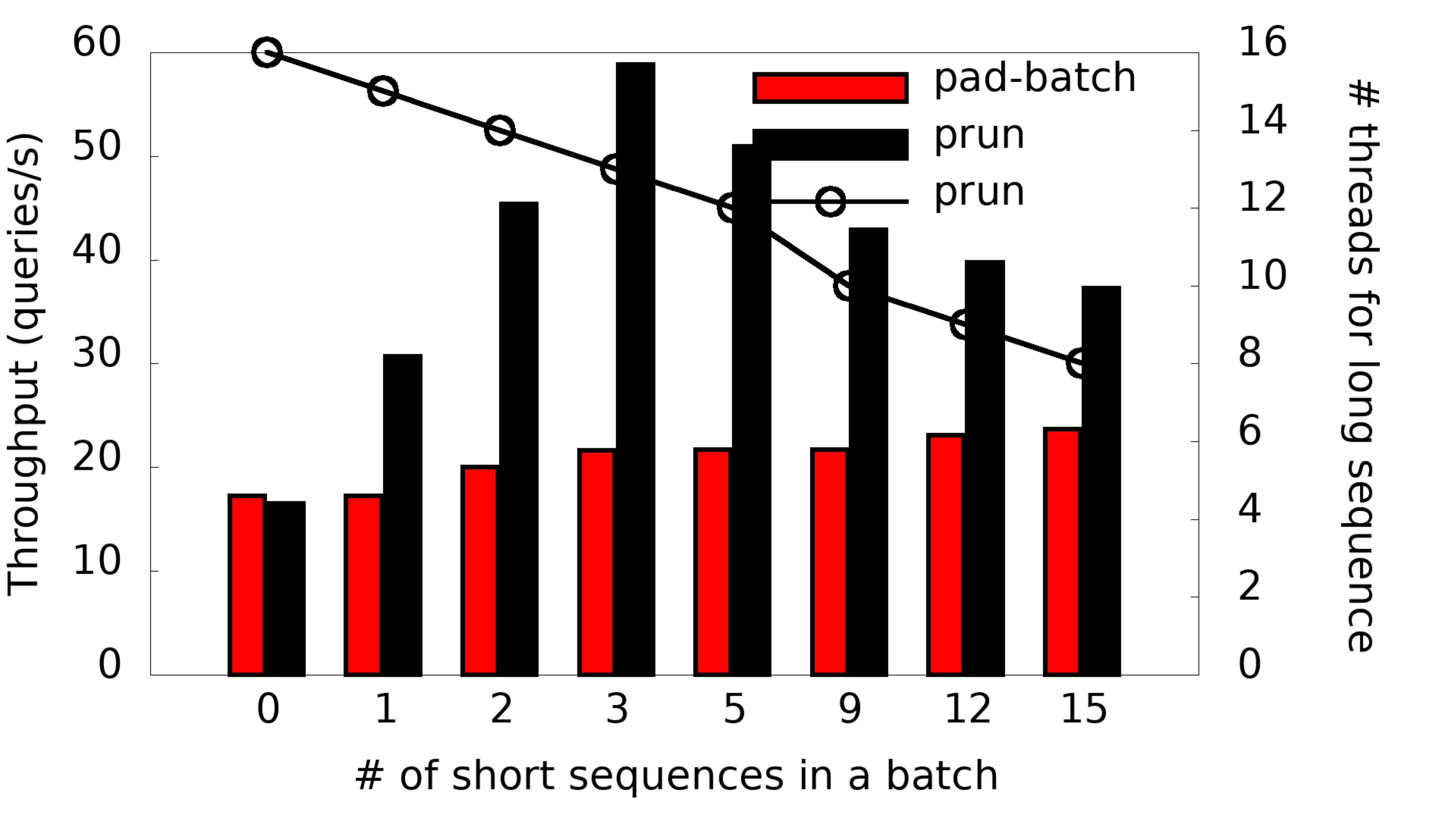


DACP evaluation: PaddleOCR / End-to-End Inference



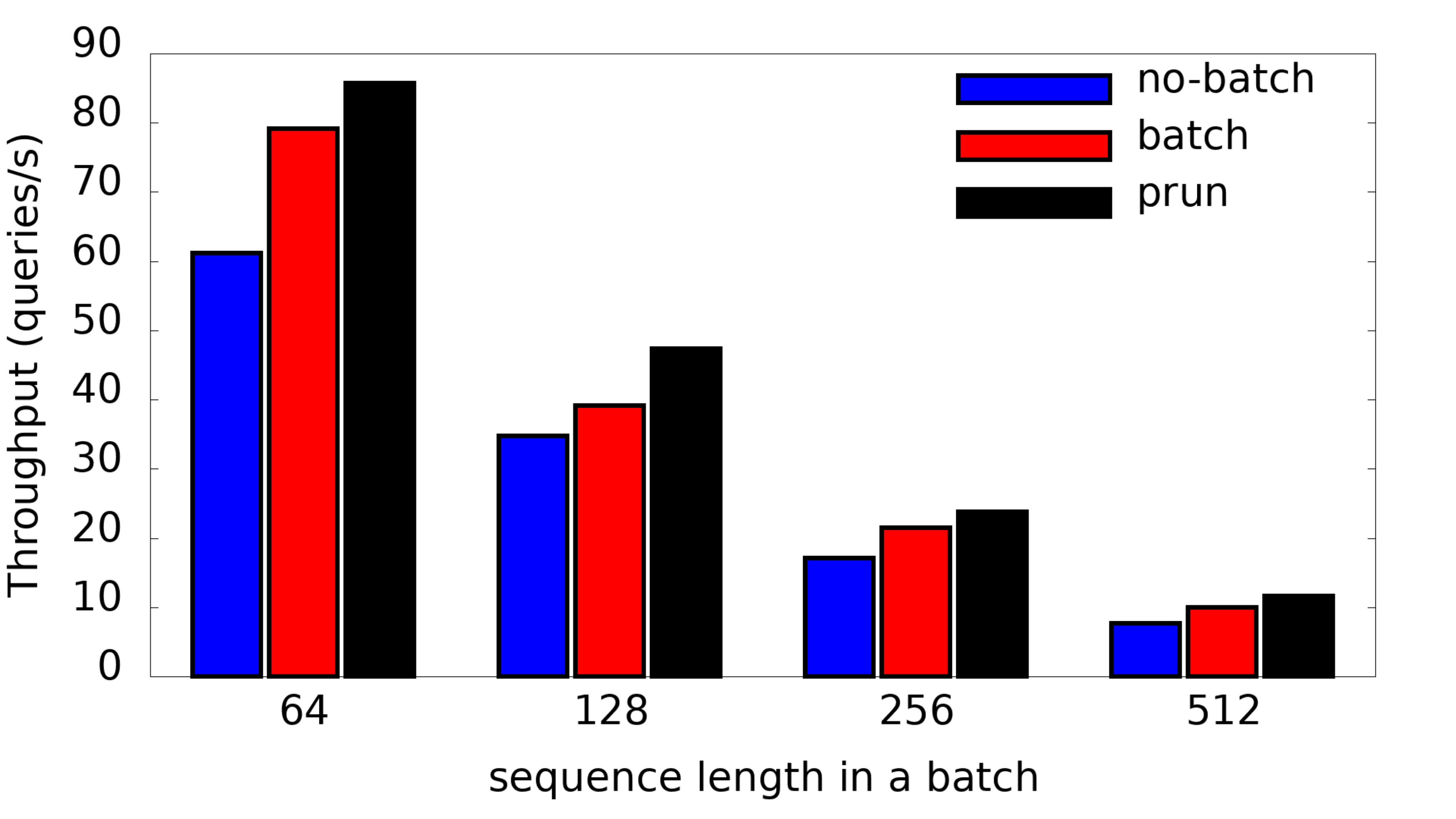
DACP evaluation: Transformers

Batching of Heterogeneous Inputs



DACP evaluation: Transformers

Batching of Homogeneous Inputs



Summary

- ML frameworks are optimized for large batches with long inputs
  - but batches are small and inputs are short during inference

- Optimize / reimplement the model or the framework

or

- Use DACP!
  - process input “chunks” in parallel (vs. processing the entire input with all available resources)
  - over 2x latency and throughput improvement
  - only minor user code changes
    - future work: apply DACP w/o user code changes
  - more details: https://arxiv.org/abs/2301.05099

Thank you!
Any questions?