Improving Inference Performance of Machine Learning with the Divide-and-Conquer Principle

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Abstract

Many popular machine learning models scale poorly when deployed on CPUs. In this paper we explore the reasons why and propose a simple, yet effective approach based on the well-known Divide-and-Conquer Principle to tackle this problem of great practical importance. Given an inference job, instead of using all available computing resources (i.e., CPU cores) for running it, the idea is to break the job into independent parts that can be executed in parallel, each with the number of cores according to its expected computational cost. We implement this idea in the popular OnnxRuntime framework and evaluate its effectiveness with several use cases, including the well-known models for optical character recognition (PaddleOCR) and natural language processing (BERT).

1 Introduction

We live in the era of unprecedented attention to machine learning (ML) from researchers and practitioners alike. New ML models across a variety of domains (or modalities, such as video, images and text) are proposed nearly daily, the models grow bigger and more sophisticated, and their components are continuously revised to achieve better accuracy scores on various tasks. While lots of attention is given to training efficiency and prediction accuracy, seemingly less effort is focused on making sure those models perform well when deployed in practice, i.e., during inference [11]. As we demonstrate in this paper, some models scale poorly (and at times, even worse!) when the number of available cores in a CPU-based deployment is increased.

Why does not the inference on CPUs scale? There are a variety of reasons, and we devote the entire section of this paper to look into some of them. Briefly, they range from the micro-level, such as the use of non-scalable operators inside ML architectures, to macro-level, such as employing ML architectures that process input iteratively.

To mitigate those scalability challenges, one might consider redesigning their ML architecture or reimplementing its non-scalable operations with a more efficient version. Such approaches, however, require either substantial ML domain specific expertise, exceptional engineering skills and familiarity with ML frameworks used for inference, significant investments (e.g., to retrain a new model, with a potential risk to the accuracy metrics), or all of the above.

In this paper, we take a different approach and propose to leverage the poor scalability of ML models by applying the Divide-and-Conquer Principle, a well-known algorithm design technique in Computer Science [8]. Specifically, instead of allocating all available computing resources (CPU cores) to the entire problem, we propose to divide the problem into smaller chunks[1], let the framework decide how the computing resources should be allocated among those chunks and then run their respective computations in parallel. We argue that in many use cases, such a division is natural and requires only trivial changes in the user code. We also describe a simple mechanism that allocates computing resources based on the expected computational intensity (or weight) of each chunk.

Consider, for instance, a model for solving a natural language processing (NLP) task such as tweet classification. Our approach allows efficient batching of inference requests of various sizes, eliminating the need for padding (a common, but wasteful solution to deal with batches of requests of variable size) and letting the framework allocate computing resources proportionally to the length of each sequence. We implement the aforementioned allocation mechanism in OnnxRuntime [24], a popular framework for training and inferencing ML models, and extend its inference API to allow user code to invoke parallel inference on multiple inputs. We demonstrate the effectiveness of our approach with several use cases, including highly popular models for image processing (PaddleOCR [14]) and NLP tasks (BERT [10]).

The remainder of this paper is organized as following. In Section 2 we elaborate on various reasons for why the inference (on CPUs) commonly does not scale well. Next, we describe in Section 3 the concept and implementation details of the Divide-and-Conquer Principle as it applies to inference. Following that, we present in Section 4 several use cases of ML models where this principle can be applied, along with the performance evaluation of its benefits. We discuss related work in Section 5 and conclude the paper in Section 6.

[1]We note that unlike the classical Divide-and-Conquer Principle [8], we divide the problem only once, although it might be possible in some cases to divide it recursively into increasingly smaller chunks that can be executed by one thread each.

2 Why is Inference Slow?

There are numerous reasons for this lack of scalability. In this section we survey some of them.

2.1 Not "enough" work

One reason is simply because the amount of computation required by a model during inference is not "enough" for efficient parallelization. As noted by Aminabadi et al. [3], kernel implementations of various ML operations are often geared towards training, which tends to consist of sizable batches of large inputs (e.g., sentences of 512 tokens). During inference, however, the batches tend to be much smaller, and often include just one input (e.g., for real-time / interactive inference). Besides, the inputs themselves can be small, e.g., a tweet or chatbot interaction consisting of just a few words.

Consider, for instance, highly popular Transformer-based [28] models for NLP tasks, such as BERT [10] or GPT-3 [5], which rely mostly (but not solely) on matrix multiplication primitives. Those primitives are known to scale well for large matrices [11, 22, 30]. However, when the actual input to the model during inference is short, matrix multiplications involve smaller and therefore, less amendable to efficient parallelization, matrices [11, 23, 30].

2.2 Non-Scalable Operators

Another reason for poor scalability of some ML models is the use of non-scalable (and often, sequential) operators in their architecture. Typically, the overhead of those operators would be negligible compared to other, more scalable parts of the model. Yet, as the number of cores increases and following directly from the Amdahl's Law [2], their negative impact of non-scalable operators on the overall inference performance would grow. Considering again the Transformer-based [28] models mentioned above, Dice and Kogan have observed that while matrix multiplication scales well, at least for long inputs, other operations such as layer normalization and softmax do not, contributing to the overall poor scalability of those models [11]. In this paper, we consider a vision-based model, which employs sequentially implemented functions for internal format conversions, which similarly cause the entire model not to scale.

We note that some of those cases could be considered a performance bug in the underlying ML framework, which could be fixed by reimplementing the respective operators with more efficient (and parallel) alternatives. This, however, requires lots of engineering effort, which includes performance analysis and deep understanding of corresponding framework implementation details. Besides, some of the ML operators, such as layer normalization [4], require careful coordination among computing threads (e.g., to compute variance and standard deviation of all the hidden units in a layer and then use those statistics to normalize the values of the units) and therefore do not lend themselves naturally for efficient parallelization.

2.3 Framework Overhead

Somewhat related to the prior point, an ML framework might add small but measurable overhead in invoking model operations. Most popular ML frameworks, such as PyTorch, Tensorflow or OnnxRuntime, support multiple backends for executing ML operations, targeting different hardware architectures (CPU, GPU, TPU), utilizing different BLAS libraries (MKL, OpenBLAS, oneDNN, etc.), different threading infrastructure (Intel TBB, pthreads, custom implementation, etc.), etc. Dispatching appropriate kernel (implementation) for every operator is efficient, but is sequential and requires non-trivial amount of work, especially when the model is executed *interactively* [11] (the default execution mode in PyTorch). This overhead becomes substantial as the actual execution time of the kernels reduces with the increased number of cores.

In addition to the above, various kernels might require specific memory layout for its input parameters (tensors), and the framework would add appropriate dummy operators for input/output conversion or data preparation [30]. As we demonstrate later in this paper, these operators might add substantial overhead as well.

2.4 Model Architecture

Quite often the high-level architecture of an ML model itself plays a substantial role in causing inference not to scale. For instance, some ML models, especially ones built for video and image processing (e.g., [14, 21, 29]), are composed as a multi-phase pipeline. The first phase of the pipeline would typically identify the points of interest in the input (e.g., text boxes in an image or a moving object in a video), while subsequent phases would process those points (iteratively or as a batch) to solve the predefined problem (e.g., identify text in the boxes or classify the moving object in the video). The inference latency of such models might grow linearly with the number of objects identified in the first phase. Furthermore, if even one phase of the pipeline does not scale well, the scalability of the entire pipeline is impaired.

2.5 Padding

Batching multiple inputs and processing them at once is a well-known way of improving inference throughput [1, 3, 9, 15, 32]. In fact, multiple serving system for machine learning models (such as TensorFlow Serving [16] or TorchServe [7]) include tunable parameters that configure how long an inference server can wait in order to batch as many input requests as possible. However, when inputs in a batch do not have exactly the same shape, they need to be padded to be processed efficiently, since underlying kernels typically anticipate batches of homogeneous inputs. The padding leads

to reduced computational efficiency, since it is treated by kernels as the rest of the input, even though the corresponding output produced by the model is dismissed.

3 Divide-and-Conquer Principle Applied to Inference

In this section, we describe the application of the Divide-and-Conquer Principle [8] to the inference of ML models at the conceptual level and as a concrete realization by implementing it in the OnnxRuntime framework. We note that applying this principle does not directly address the reasons for poor scalability detailed in the previous section. In fact, the advantage of our approach is that one does not have to identify and/or fix any scalability bottlenecks in their models to rip the benefits of its underlying idea.

3.1 Concept

The basic idea is pretty straightforward — consider a computation job J, which can be broken into k independent parts, j_1, j_2, ..., j_k, which can be executed in parallel. Assume we have an oracle assigning relative weight $w_i \in (0, 1]$ corresponding to, e.g., the number of required floating point operations (FLOPs) or single-thread latency of the computation job part j_i. Finally, assume we have C computing cores available. We strive to allocate to each part the number of cores relative to its weight, namely, we assign $c_i = max\{1, \lfloor w_i * C \rfloor\}$ cores for the part j_i. This effectively means allocating c_i worker threads for j_i since we later create one worker thread per core (as common in ML frameworks, including in OnnxRuntime).

Note that $\sum_{i=1}^{k} c_i$ might be larger than C. This is obvious when the number of job parts, k, is larger than C, but it is possible even when $k \leq C$. This does not create a problem other than implying that some job parts will be run after other job parts have finished (rather than running them all in parallel). At the same time, due to the rounding-down (floor) function intended to reduce the above possibility of oversubscription, some unallocated cores might remain. To avoid this waste of available resources, we sort all the job parts by their remaining unallocated weight, i.e., by $w_i * C - \lfloor w_i * C \rfloor$, and assign one core to each part in the descending order, up until all cores are allocated. The C++-like pseudocode for the entire algorithm is given in Listing 1.

Naturally, the idea described above raises the question of how to assign relative weight to a job part j_i. In general, this can be done with the help of a profiling phase and a lightweight classification mechanism, which associates job parts of the same (or similar) shape (as the one encountered during the profiling phase) to the relative weight obtained during profiling. However, in all our cases considered in Section 4, the weight is simply set proportionally to the size of input tensors. Specifically, let s_i be the size of the input tensor for job part j_i. We set w_i to $\frac{s_i}{\sum_{i=1}^{k} s_i}$. This makes sense since for our use cases, the amount of computation (expressed as the number of required FLOPs) grows roughly linearly with the input tensors' size.

```
1  vector<int> allocate(vector<Tensor> inputs, int numCores) {
2    vector<int> threadAllocation;
3    vector<tuple<int, float>> threadUnallocatedWeight;

4    int numInputs = inputs.size();
5    int allocatedCores = 0;
6    int index = 0;

7    int totalSize = 0;
8    for (auto j_i : inputs) totalSize += j_i.size()

9    for (auto j_i : inputs) {
10     int numThreadsToUse = 1;

11     if (numInputs <= numCores) {
12       int size = j_i.size();
13       float w_i = ((float)size) / totalSize;
14       int numThreadsToUse = floor(w_i * numCores);

15       unallocatedWeight.add(
16          make_tuple(index, w_i * numCores - numThreadsToUse);
17     }

18     // this may happen due to flooring
19     if (numThreadsToUse < 1) numThreadsToUse = 1;

20     threadAllocation.add(numThreadsToUse);

21     allocatedCores += numThreadsToUse;
22     index++;
23   }

24   if (allocatedCores < numCores) {
25     // sort the vector in decreasing order by
26     // comparing the second field in each tuple
27     sort(unallocatedWeight, bySecondField);

28     int nextToAdjust = 0;
29     while (allocatedCores < numCores) {
30       // fetch the first field in the `nextToAdjust` tuple
31       index =
32          unallocatedWeight[nextToAdjust % numInputs].get(0);
33       threadAllocation[index]++;
34       allocatedCores++;
35       nextToAdjust++;
36     }
37   }
38   return threadAllocation;
39 }
```

Listing 1. Thread allocation algorithm

```
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

3.2 Implementation Details

We extend the API of the `InferenceSession` class of OnnxRuntime with a new `prun` method. This method is modeled after the existing `run` method used as the main entry point when running inference. The main difference is that `prun` accepts a list of inputs (instead of just one) and returns a list of outputs.

Internally, the implementation of `prun` iterates over the list of inputs, calculates their size (after validating those are tensors) and corresponding relative weight, and applies the allocation algorithm described in Listing 1 to associate the number of worker threads with each input (job part). Following that, the implementation creates one worker thread for each input, and runs them in parallel. Each worker thread, in turn, creates a thread pool of the size calculated by the allocation algorithm (the thread pool includes the worker thread itself), and invokes the `run` method of the `InferenceSession` object with that thread pool. The entire patch of the OnnxRuntime codebase to implement the `prun` functionality and other minor internal changes (such as having the `run` method to accept a thread pool as an optional argument instead of always using the default pool) consisted of around 200 lines of code.

On the user side, the code also has to change to make use of the new `prun` API. Those changes, however, are quite straightforward. Instead of invoking `run` for every job, a user needs to create a list of job parts and call `prun`. In addition, the user needs to rearrange the post-processing code to iterate over the results of `prun`, and apply any post-processing to each returned output (object). As an example of what the user code changes entail, we show the original Python code (edited for brevity and clarity) of the TextRecognizer class in PaddleOCR (Listing 2) alongside the modified version that makes use of the new `prun` API (Listing 3).

4 Use Cases

Before we detail the use cases where the Divide-and-Conquer Principle is beneficial and report on our performance findings, we give a brief summary of our evaluation setup and methodology. We run all our experiments on a 16-core AMD-based VM in Oracle Cloud (aka OCI VM.Standard.E3.Flex). (We also ran some experiments on a newer E4 shape, but have not noticed substantial differences). To reduce performance variability, especially as we create separate thread pools for the variants that use `prun`, we use thread binding (pinning), for all the evaluated variants. Every experiment was repeated 5 times, and we report the mean. We note that the standard deviation of all reported results, except for one specific case discussed below, was extremely low (typically, less than 1% of the mean). For our experiments, we use the latest release versions (as of the date of writing this paper) of the corresponding software, specifically OnnxRuntime v1.11.1 and PaddleOCR v2.5.

4.1 Sequential Pipeline

Our first example of where applying the Divide-and-Conquer Principle is extremely useful is PaddleOCR [14]. PaddleOCR is a lightweight OCR system, which consists of three parts: Text Detection, Text Classification (called Detection Boxes

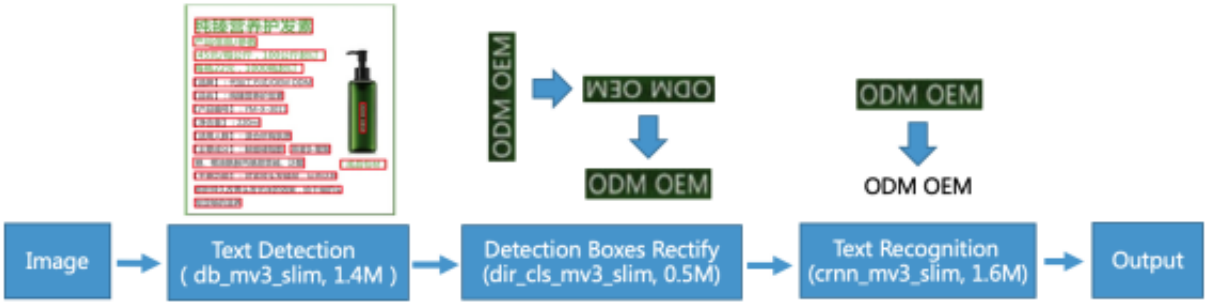


Figure 1. PaddleOCR 3-phase pipeline (edited version of Figure 2 from [14]).

Rectify in [14]) and Text Recognition. Each of those parts corresponds to a separate ML model.

The OCR pipeline accepts an image file and passes it first through the text detection phase whose objective is to locate text areas in the image. The output of this phase is a list of potential text boxes' coordinates. Next, the list is iterated over, and each item in that list (i.e., a text box) is sent to the text classification model, which decides whether the box needs to be transformed into a horizontal rectangle box before the actual text recognition takes place. Based on the classifier's decision, each box is altered respectively. Finally, the list is iterated over again, and each item is sent to the text recognition model for inference, which recognizes the text in the given box and produces the actual character sequence based on the supplied character dictionary. This process is depicted in Figure 1, which is a redacted version of Figure 2 from [14].

In our experiments with PaddleOCR, we observe that the system does not scale well with the increase in the number of available cores. We demonstrate that in Figure 2 depicting inference latency as a function of available cores (which directly translates into the number of worker threads used by the runtime). For all experiments in this section, including the one in Figure 2, we use a subset of images from the OpenImages dataset [17], selected according to a criterium described below.

In Figure 2, we break the total latency into time spans corresponding to the three phases of the OCR pipeline discussed above. As one can notice, the average inference latency goes down from 517 ms for 1 thread to 358 ms for 4 cores and then back up to 416 ms for 16 cores. Interestingly, the Text Classification phase shows negative scalability, where it takes 25 ms to process an image, on average, with 1 thread, but it takes 35 ms to do the same with 16 threads — a slowdown of 1.4x. This shows an example of a system where, beyond a certain point, adding more threads not only does not help, but actually harms performance. Discussing concrete reasons for the lack of scalability of these specific models is not in the scope of this paper. For a curious reader, however, we note that a built-in OnnxRuntime profiling tool shows inflated execution times for the output reordering operators (which are inserted by the framework, along with the input reordering operator, to convert the memory layouts of input arguments for various kernels).

We apply the Divide-and-Conquer Principle to the last two phases of the OCR pipeline, namely the Text Classification and Recognition. To that end, instead of invoking the corresponding models for each text box produced by Text Detection, we send all the boxes to the runtime (by invoking the `prun` API) and effectively let the runtime decide how many cores / worker threads to allocate each box based on its relative size. The required changes to implement this functionality in the Text Recognition phase are depicted in Listing 3; the changes to the Text Classification phase are similar.

For our performance evaluation, we compare the `prun` implementation as discussed in Section 3 (and depicted in Listing 1), which we denote as `prun-def` on the charts, to a few simple variants. The first variant, denoted as `prun-1`, simply allocates one worker thread to each input in the list given to `prun`. The second variant, denoted as `prun-eq`, allocates an equal number of cores for each input (but at least one), i.e., sets $c_i = max\{1, \lfloor k/C \rfloor\}$. Our motivation is to show that trivial solutions might also be useful in certain scenarios (as discussed below), yet they tend to underperform compared to `prun-def`.

We note that the benefit of `prun` in this use case is possible only when there are at least two text boxes identified in the Text Detection phase. Otherwise, the other two phases would not be used (if no text boxes detected) or the `prun-def` variant will use the same (maximum) number of cores as the base (unmodified) version (if only one text box is detected). As a result, the subset of images used for performance evaluation in this section includes images with at least two identified text boxes. The pie chart in Figure 3 shows the distribution of the actual number of boxes detected in the first phase of the OCR pipeline for the entire dataset. The total number of images in the dataset was 500 – this number was chosen to keep the evaluation times reasonably short. (We note that we also ran evaluations on a larger dataset that includes images

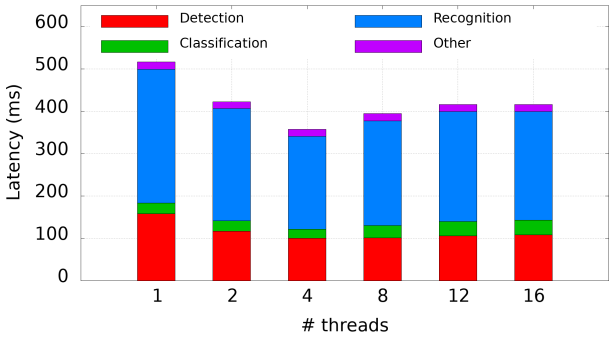


Figure 2. Inference latency of PaddleOCR with a varying number of threads, broken down by the three phases of the pipeline.

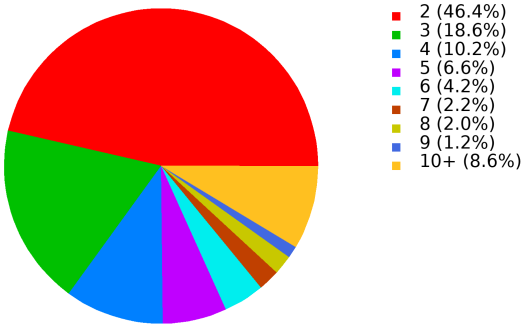


Figure 3. Distribution of the number of detected text boxes in the input dataset.

with less than two text boxes and confirmed that the use of `prun` does not create any overhead in those cases.)

In light of the discussion above, we break down the comparison of the latency results by the number of detected boxes, as depicted in Figure 4. The latency numbers in this figure were collected with 16 cores; we discuss the overall scalability trends later on. We also break down the performance in two of the phases where we have used `prun`, namely Text Classification (Figure 4 (a)) and Recognition (Figure 4 (b)).

Considering the results in Figure 4, one can notice that, as expected, the benefit of `prun` increases with the number of detected text boxes. For instance, when considering the total end-to-end latency (Figure 4 (c)), with only two boxes `prun-def` outperforms `base` by 1.26x. However, with 9 and 10+ boxes, `prun-def` outperforms `base` by 2.2x and 1.76x, respectively.

It is interesting to compare the performance of `prun-def` with other pun-based variants. As one can notice in Figure 4 (a), the `prun-1` variant produces the lowest latency when the number of detected boxes is small. In fact, the `base` variant also performs better than `prun-def` in this case. We attribute this to two factors. First, this specific phase of the pipeline shows negative scalability, which can be also seen in Figure 2. Therefore, best performance is achieved when fewer threads per box is used in this phase, which is what `prun-1` effectively achieves. Second, `prun-def` (and `prun-eq`) create and destroy more threads than `prun-1` in those cases as they create thread pools containing more threads for each `prun` invocation. This adds small, but non-negligible overhead given that the the execution time of this phase is short. In the future work, we intend to experiment with reusing thread pools between `prun` invocations. As the number of detected boxes increases, however, all `prun` variants allocate less threads (or even just 1) per each box, and they allocate a similar number of threads for their pools, thus closing the gap with the `prun-1` variant.

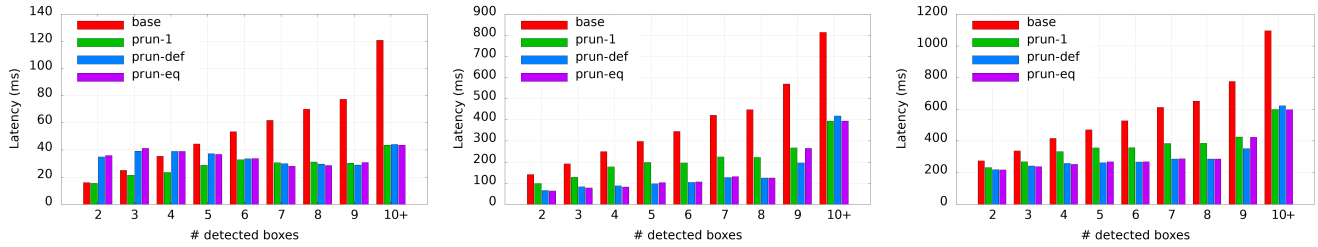
When the Text Recognition phase is concerned (cf. Figure 4 (b)), however, it is apparent from Figure 2 that one can improve its latency by using more than one thread. We note that, quantitively, this phase is also far more dominant than the Text Classification one. Here, `prun-def` manages to achieve best or close to best result across all counts of detected boxes, which translates to overall highly competitive end-to-end inference performance (cf. Figure 4 (c)). In general, the results in Figure 4 call for a dynamic mechanism, which would choose the best thread allocation strategy based on the given workload and available resources. Devising and experimenting with such a strategy is left for future work.

Finally, we shed more light on how the scalability improves with the use of `prun` in Figure 5, where we vary the number of cores (and therefore, the total number of worker threads) available for OnnxRuntime. Once again, we include the latency of each of the two last phases of PaddleOCR (denoted as Rec for Text Recognition and Cls for Text Classification) along with the end-to-end (Total) latency. We include only the results of the `base` and `prun-def` variants (denoted simply as `prun` in Figure 5), for clarity.

Overall, one can notice similar trends to the ones discussed above. In the `base` version, the Text Recognition phase does scale up to 4 threads, but then its performance suffers as the number of threads increases. The `prun` variant avoids this performance degradation, and in fact, continues to scale up to 16 threads. Indeed, when considering the Text Recognition phase only, the `prun` variant outperforms `base` by more than 2.3x at 16 threads. However, since both variants have an identical Text Detection phase, which according to Figure 2 subsumes a substantial part of the total latency, the end-to-end speedup of `prun` is only 1.5x at 16 threads.

4.2 Batching of Heterogeneous Inputs

Our next example concerns with the Transformer architecture [28], which revolutionized the domain of NLP when it was introduced in 2017 and has been applied to other domains since then (e.g., [12, 18]). This architecture consists



(a) Text Classification **(b)** Text Recognition **(c)** End-to-End Inference

Figure 4. The impact of using `prun` in PaddleOCR.

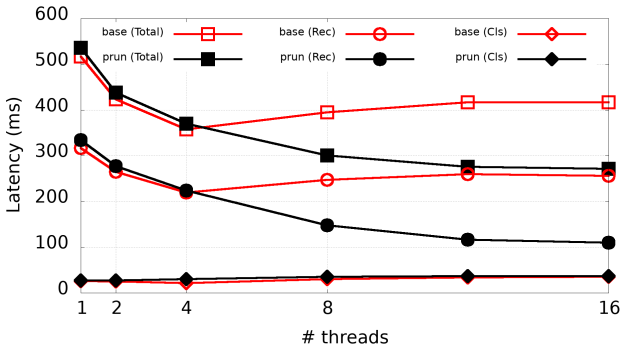


Figure 5. Total (end-to-end) inference latency of PaddleOCR with a varying number of threads. Also shown the latency of Text Classification (Cls) and Text Recognition (Rec) phases

of a stack of layers, each composed of a self-attention block followed by a fully connected network [28]. Past work has shown that the majority of computation cycles in Transformers is spent on (scalable) matrix multiplication operations, yet up to one third of the cycles is spent elsewhere (i.e., less scalable operations) [11].

It is well-known that one way to improve the inference performance (specifically, throughput) of Transformers is through *input batching* [3, 15, 30]. This strategy works well, however, when the inputs have the same length. Otherwise, one has either give up on batching, or pad inputs to the same length. The latter results in wasted computation cycles, since special padding tokens are treated exactly as input tokens by the architecture and dismissed at the end of the computation.

This situation presents an ideal case for applying the Divide-and-Conquer Principle. Instead of padding the inputs of various lengths up to the longest input in the batch, we can run inference on those inputs (as they are, without padding) using the `prun` API, and let the runtime decide how many cores should be used to process each of the inputs. We modify the Transformer benchmark built into the OnnxRuntime [25] to implement this strategy.

To evaluate the effectiveness of the approach described above, we set up an experiment where we generate X inputs of a length chosen uniformly and randomly out of the range $[16, 512]$. We then compare the `pad-batch` version in which all X inputs are padded to the longest length in the given batch with the `prun` version in which the inference is invoked with `prun` on all inputs in the batch. We show results with the highly popular BERT model [10] (specifically, "`bert-based-uncased`"). We have also experimented with other Transformer-based models (such as "bert-large-uncased" or "roberta-base") measuring similar qualitative results.

We note that this experiment includes inherent amount of randomness — a batch of small sentences is as likely to be chosen as a batch of long sentences. In an attempt to reduce the anticpated high varaince of the results, we opted to repeat the experiment 1000 times, and so for each X, each data point is an average of 1000 results. Figure 6 presents the throughput results with batches of various sizes (i.e., X varies from 2 to 8), with error bars depicting the standard deviation of the reported mean. Even though `prun` outperforms the `pad-batch` variant across all batch sizes, the variance in the measured results remains exceptionally high.

As a result, we setup two additional experiments in a more controlled way likely to produce more stable results. In the first, we simply preset the lengths of various sequences in each batch. For instance, a batch denoted as "16-64-256" includes three sentences, one is 16, another is 64 and yet another is 256 tokens long. We show the results of this experiment in Figure 7. Here, the `prun` version easily outperforms the `pad-batch` variant, which has to pad all sequences to the longest sequence in a batch. As one might expect, the benefit from using `prun` increases with the number of sentences in a batch, as this variant eliminates all the redundant work associated with padding.

In the second experiment, we use a batch of 1 long sentence (256 tokens long) and X short sequences of 16 tokens each, where we vary X between 0 and 15. We show the throughput results of this experiment in Figure 7, along with a curve depicting the number of threads allocated by `prun` for the long sequence in the batch.

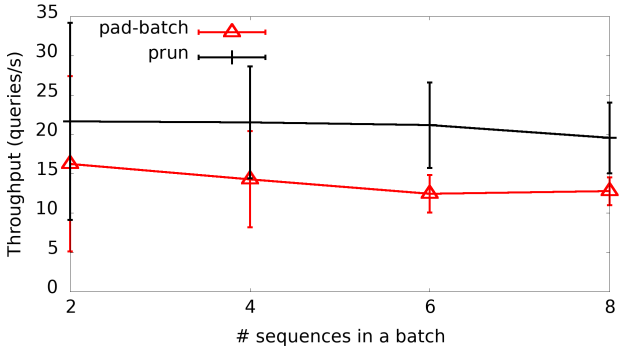


Figure 6. Throughput of inferencing BERT on batches of sequences of sizes chosen randomly from the range [16, 512]

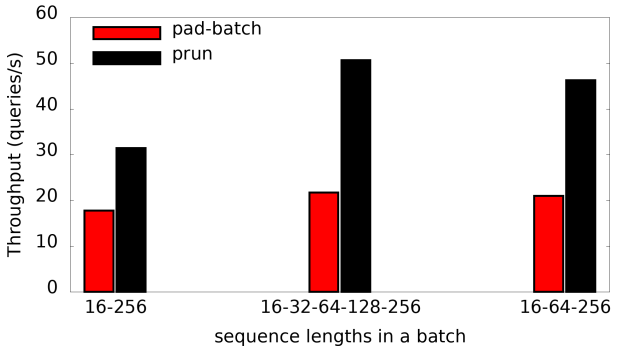


Figure 7. Throughput of inferencing BERT on batches of sequences of various preset sizes

There are several interesting observations that can be made here. First, when X=0, i.e., the batch contains only one long sentence, both variants employ all available cores to process that batch, producing similar result. This shows that the overhead of using `prun` when the input has only one chunk is negligible. Second, the throughput of the `pad-batch` version grows, but modestly with the increase in the number of short sequences. This is because, as stated above, a larger batch of (padded) sequences helps to achieve better throughput with Transformers. At the same time, the throughput growth with `prun` is much more dramatic up to 3 short sequences in a batch and then it declines, but stays well above that achieved with `pad-batch`. Both phenomena can be explained with the fact that inferencing a sequence of 256 tokens takes about the same time with 16 threads as it takes with 13. Thus, adding a few short sequences into the batch, each allocated with just 1 thread (as they have small relative weight), has negligible impact on the latency, but improves throughput. With more short sequences in a batch, less threads are allocated for the long sequence (as can be seen in Figure 7) and its inference latency grows. This causes the overall throughput to decrease.

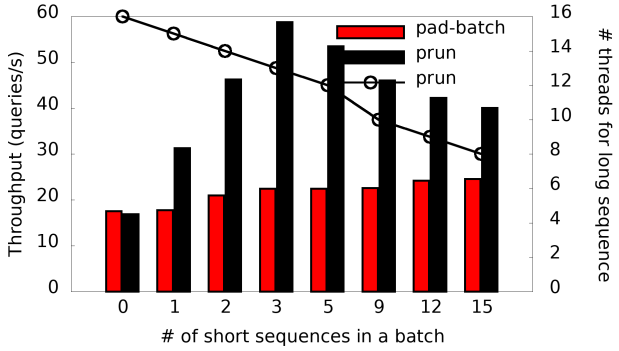


Figure 8. Throughput of inferencing BERT on a batch containing one long sentence of 256 tokens and X short sequence with 16 tokens each, where X varies 0 to 15. In addition, we show how many threads are dedicated to the inference of the one long sentence in the batch in the `prun` variant

4.3 Batching of Homogeneous Inputs

Our last example follows directly from the discussion in Section 2 on the lack of scalability in ML models. As already mentioned, while Transformers models heavily use scalable matrix multiplication operations, they also employ less scalable operations. The impact of the latter grows with the increase in the number of cores. Therefore, one may benefit form the Divide-and-Conquer Principle applied to Transformers *even when the batch includes inputs of the same length.*

As a concrete example, consider a batch of two inputs. Instead of using all available cores to process the batch, we will use half the cores for each input. Intuitively, the less scalable operators create less relative overhead when less cores are used and the input sequence is shorter (i.e., contains half the tokens compared to the entire batch).

Figure 9 demonstrates this effect with batches of inputs of equal lengths. In addition to the `pad-batch` variant (which we simply call `batch` here, as no padding is required) and `prun`, we include a `no-batch` variant, which runs inference on each sequence in a given batch one at a time. Note that we include the latter to simply demonstrate the benefits of batching in general, confirming previous findings [3, 15, 30]. Each set of bars in Figure 9 corresponds to a batch of 4 sentences with the given length (from 64 tokens to 512). Overall, the `prun` version yields a more modest (yet non-trivial) speedup over `batch` compared to the case of non-homogeneous inputs in Section 4.2. This is expected, since in this case the room for improvement (over `batch`) does not include wasted computation related to padding.

5 Related Work

As mentioned in the Introduction, the major focus of the ML community has been on improving the accuracy and training performance of proposed models, while efficient

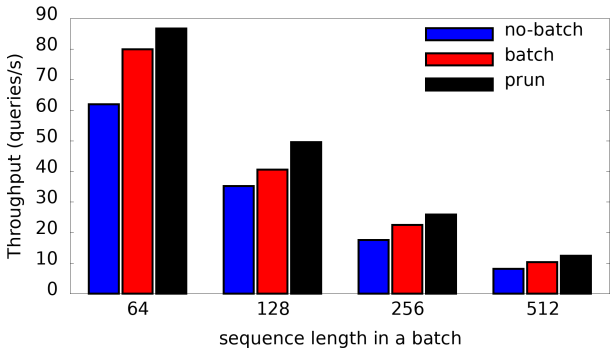


Figure 9. Throughput of inferencing BERT with batches of 4 sequences of equal size

inferencing and serving of those models receives relatively less attention. Yet, there have been some notable exceptions of work focused specifically on inference performance, and we survey the most relevant results hereafter. As an aside, we note that many of the results below come from less formal blog posts published by various companies, highlighting the great practical importance of efficient inference.

Wang et al. [30] explore various factors that influence inference performance in TensorFlow, including the choice of a specific math library, a thread pool library, availability of SIMD (single instruction multiple data) support, etc. They identify data preparation as one of the causes for poor scalability of small matrix multiplication operations, something we more generally attribute to framework overhead in Section 2. They come up with a set of guidelines one can use to tune TensorFlow settings to achieve better performance compared to the one achieved with settings recommended by TensorFlow authors or Intel.

With the tremendous rise in popularity of Transformers, several papers and blog posts focus on its inference performance. Dice and Kogan investigate inference performance of Transformers on CPUs [11]. Their analysis shows that most inference computation cycles are spent in matrix multiplication operations. Hence, they propose an adaptive matrix multiplication optimization aimed at reducing the latency of those operations and subsequently improving the overall inference performance. Intel engineers describe an effort to optimize inference of BERT in Apache MXNet using the GluonNLP toolkit, where one of the ideas is to quantize the model for better performance with lower precision [31]. Similar quantization ideas (along with *distillation*, another common method of reducing the size of a model [27]) were employed by Roblox to speedup their deployment of BERT on CPUs [19]. The same blog post also mentions that eliminating padding of input sentences has led to better performance (though the authors did that for batches of 1 input only). A Microsoft team [26] describes their effort on accelerating BERT with OnnxRuntime through operation fusion that helps to reduce the amount of overhead (e.g., memory copying) in invoking each kernel individually.

A few recent papers and projects have looked into the deficiency of padding of heterogenous inputs. Fang et al. [15] propose a sequence-length-aware batch scheduler, which aims to batch requests of a similar size, thus reducing the cost of zero padding of all requests into one batch. It requires a profiling phase during which the inference cost of various batches is collected. Du et al. [13] propose to carefully redesign the GPU kernels employed by Transformers to eliminate most redundant computation associated with zero padding. The Effective Transformer project by ByteDance [6] aims to dynamically remove and restore padding during different calculation stages. All those efforts target specifically the inferencing Transformers on GPUs, and it is not clear how efficient they would be on CPUs and/or with other architectures.

Beyond Transformers, Liu at et. [20] describe NeoCPU, an approach for optimizing CNN inference on CPUs. NeoCPU proposes a configurable design of an efficient convolution operation that can be tuned efficiently to popular CPUs. This design is coupled with a scheme for obtaining the best memory layout for data in different operations of a CNN model, in order to minimize the overhead of transforming the data between various individual operations.

6 Discussion

In this paper, we have discussed various reasons for the lack of scalability of inferencing ML models. While the reasons vary from micro to macro-levels, the common motive is that existing ML frameworks are geared towards high performance training. This is expressed by the fact that kernels for common operations are typically optimized for large batches with long inputs, ignoring relatively small overheads in various parts of those frameworks that are immaterial to the overall training performance. However, during inference the batches tend to be much smaller and contain shorter inputs, thus making those overheads more prominent. A somewhat similar observation has been made by Aminabadi et al. [3].

We leverage this poor scalability and describe a simple, yet powerful approach, in which the given input is broken into chunks and each chunk is processed in parallel, instead of using all available resources for the entire input. As we demonstrate with a few well-known models, this approach improves inference scalability and ultimately can lead to over 2x latency and throughput improvements.

This work offers several directions for future research. First, we want to explore more dynamic thread allocation strategies, e.g., ones that can better adjust to the cases where the weight of a work chunk does not correlate linearly with its size and/or where the underlying model performs best while running with a single thread. Second, we want to find ways to automate splitting the input into chunks that can

be processed in parallel, lowering the cost (in terms of user code changes) of using `prun` even further. Finally, we want to explore other use cases where the use of `prun` would be beneficial, including other ML models that feature a pipeline-based architecture (e.g., [21, 29]).

Acknowledgments

The author would like to thank Dave Dice for valuable comments on an early draft of this paper.

References

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (AFIPS)*, page 483–485, 1967.

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. DeepSpeed inference: Enabling efficient inference of transformer models at unprecedented scale. *CoRR*, abs/2207.00032, 2022.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[6] ByteDance. Effective Transformer. https://github.com/bytedance/effective_transformer. Accessed: 07-29-22.

[7] PyTorch Serve Contributors. TorchServe. https://pytorch.org/serve. Accessed: 07-28-22.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[9] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 613–627, 2017.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (NAACL-HLT)*, pages 4171–4186, 2019.

[11] Dave Dice and Alex Kogan. Optimizing inference performance of Transformers on CPUs. In *Workshop on Machine Learning and Systems (EuroMLSys)*, 2021.

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*, 2021.

[13] Jiangsu Du, Jiazhi Jiang, Yang You, Dan Huang, and Yutong Lu. Handling heavy-tailed input of transformer inference on gpus. In *ACM International Conference on Supercomputing (ICS)*, 2022.

[14] Yuning Du, Chenxia Li, Ruoyu Guo, Xiaoting Yin, Weiwei Liu, Jun Zhou, Yifan Bai, Zilin Yu, Yehua Yang, Qingqing Dang, and Haoshuang Wang. PP-OCR: A practical ultra lightweight OCR system. *CoRR*, abs/2009.09941, 2020.

[15] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: an efficient GPU serving system for transformer models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 389–402, 2021.

[16] Google. Tensorflow Serving. https://www.tensorflow.org/tfx/guide/serving. Accessed: 07-28-22.

[17] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017.

[18] Hang Le, Juan Miguel Pino, Changhan Wang, Jiatao Gu, Didier Schwab, and Laurent Besacier. Dual-decoder transformer for joint automatic speech recognition and multilingual speech translation. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, pages 3520–3533, 2020.

[19] Quoc N. Le and Kip Kaehler. How We Scaled Bert To Serve 1+ Billion Daily Requests on CPUs. https://blog.roblox.com/2020/05/scaled-bert-serve-1-billion-daily-requests-cpus. Published: 05-27-20, Accessed: 08-02-22.

[20] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on cpus. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 1025–1040, 2019.

[21] Matteo Maggioni, Yibin Huang, Cheng Li, Shuai Xiao, Zhongqian Fu, and Fenglong Song. Efficient multi-stage video denoising with recurrent spatio-temporal fusion. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 3466–3475, 2021.

[22] Bryan Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In *Proceedings of the International Conference on Parallel Processing (EuroPar)*, page 748–757, 2007.

[23] Ian Masliah, Ahmad Abdelfattah, A. Haidar, S. Tomov, Marc Baboulin, J. Falcou, and J. Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *Proceedings of the International Conference on Parallel Processing (EuroPar)*, page 659–671, 2016.

[24] Microsoft. OnnxRuntime. https://onnxruntime.ai. Accessed: 08-02-22.

[25] Microsoft. Transformer Model Optimization Tool Overview. https://github.com/microsoft/onnxruntime/tree/master/onnxruntime/python/tools/transformers. Accessed: 08-02-22.

[26] Emma Ning, Nathan Yan, Jeffrey Zhu, and Jason Li. Microsoft open sources breakthrough optimizations for transformer inference on gpu and cpu. https://cloudblogs.microsoft.com/opensource/2020/01/21/microsoft-onnx-open-source-optimizations-transformer-inference-gpu-cpu/. Published: 01-20-20, Accessed: 01-06-21.

[27] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[29] Li Wang and Dennis Sng. Deep learning algorithms with applications to video analytics for a smart city: a survey. *CoRR*, abs/1512.03131, 2015.

[30] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. Exploiting parallelism opportunities with deep learning frameworks. *ACM Trans. Archit. Code Optim.*, 18(1), 2021.

[31] Shufan Wu, Tao Lv, Pengxin Yuan, Patric Zhao, Jason Ye, and Haibin Lin. Optimization for BERT Inference Performance on

CPU. https://medium.com/apache-mxnet/optimization-for-bert-inference-performance-on-cpu-3bb2413d376c. Published: 09-12-19, Accessed: 08-02-22.

[32] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *USENIX Annual Technical Conference (ATC)*, 2022.