DOPpler: Parallel Measurement Infrastructure for Auto-tuning Deep Learning Tensor Programs

Damian Borowiec, Gingfung Yeung, Adrian Friday, Richard Harper, Peter Garraghan

Abstract—The heterogeneity of Deep Learning models, libraries, and hardware poses an important challenge for improving model inference performance. Auto-tuners address this challenge via automatic tensor program optimization towards a target-device. However, auto-tuners incur a substantial time cost to complete given their design necessitates performing tensor program candidate measurements serially within an isolated target-device to minimize latency measurement inaccuracy. In this paper we propose DOPpler, a parallel auto-tuning measurement infrastructure. DOPpler allows for considerable auto-tuning speedup over conventional approaches whilst maintaining high-quality tensor program optimization. DOPpler accelerates the auto-tuning process by proposing a parallel execution engine to efficiently execute candidate tensor programs in parallel across the CPU-host and GPU target-device, and overcomes measurement inaccuracy by introducing a high-precision on-device measurement technique when measuring tensor program kernel latency. DOPpler is designed to automatically calculate the optimal degree of parallelism to provision fast and accurate auto-tuning for different tensor programs, auto-tuners and target-devices. Experiment results show that DOPpler reduces total auto-tuning time by 50.5% on average whilst achieving optimization gains equivalent to conventional auto-tuning infrastructure.

Index Terms—Measuring Program Latency, Program Auto-tuning, Deep Learning Compilers, Deep Learning Systems

1 Introduction

Deep Learning (DL) has unquestionably become important for both industry and academia. Demand for ever better performing DL models offering accurate and low-latency inference, drives research into optimizing DL models to reduce their inference latency [1]. An effective mean to achieve this goal is to optimize individual computational components of DL models – *tensor programs* – towards specific target-device characteristics (e.g. cache & memory hierarchy, thread scheduling). Manually performing such optimizations is difficult and time-consuming, requiring expertise in DL and systems engineering. This has led to the rise of *auto-tuning*: an automated optimization of tensor programs towards a target-device, facilitated by DL compilers such as TVM [2] and Halide [3].

Auto-tuning optimizes tensor program latency on a target-device by conducting an automated implementation search for tensor programs [2], [3], [4], [5], and requires performing thousands of costly iterative measurements of candidate implementations per target-device. Such measurements must be performed in an isolated target-device serially to ensure accurate latency measurement [2]. This *serial* approach yields low platform resource utilization and throughput, meaning auto-tuning of even small DL models requires hours to achieve reasonable performance improvement [5], resulting in reduced cost-efficiency for DL providers. Given the growing industrial adoption of auto-tuning in large-scale DL deployments (Amazon SageMaker [6], Microsoft Watch For [7]) and the emergence of Optimization-as-a-Service solutions (OctoML Octomizer [8]), slow DL auto-tuning is a crucial issue to address.

• *D. Borowiec, G. Yeung, A. Friday, R. Harper and P. Garraghan are with the Lancaster University, UK. Email: {d.borowiec, g.yeung1, a.friday, r.harper, p.garraghan}@lancaster.ac.uk*

Manuscript received XX XX 2022 (corresponding author: Damian Borowiec)

One approach to overcome these issues is to provide auto-tuners the capability to measure multiple tensor program candidates simultaneously within a target-device. This appears to be an appealing solution, as computing system parallelization can improve throughput and resource efficiency. However, this has yet to be attempted for auto-tuning, as established thinking is that introducing any degree of parallelism during latency measurement will incur high inaccuracy stemming from programs (i.e. GPU kernels) competing for target-device resources. Such inaccuracy is further compounded given that all auto-tuners measure tensor program latency at the CPU-level, and not on the target-device itself. This is appropriate for measurement in isolation, however incurs measurement inaccuracy in parallel due to CPU-level contention from awaiting GPU kernels to complete execution. This presents a challenging conundrum: performing a high degree of tensor program latency measurements in parallel will speedup auto-tuning, however increases measurement inaccuracy leading to sub-optimal optimization – the purpose of auto-tuning.

In this paper we present DOPpler – a parallel DL auto-tuning measurement infrastructure. DOPpler accelerates auto-tuning measurement whilst enabling state-of-the-art auto-tuners to attain optimization performance improvements equivalent to conventional serial measurement infrastructure. DOPpler achieves this by performing tensor program latency measurements in parallel using simultaneous CPU processes and leveraging the per-process, GPU driver-level serialization, reducing the proportionally significant execution costs occurring at the CPU-level measurement harness. To maintain measurement accuracy, DOPpler performs measurements directly on target-device, instead at the CPU-level currently used. During auto-tuning, DOPpler dynamically determines an optimal parallelism level with respect to tensor program execution characteristics, to accel-

erate auto-tuning and minimize measurement inaccuracy. Specifically, the core contributions [9] of our work are:

We explore the feasibility of parallel auto-tuning measurements, identifying auto-tuner design limitations causing GPU/CPU performance bottlenecks, measurement inaccuracy, unpredictable CPU-level latency measurements, and execution timeouts.
We propose DOPpler: an auto-tuning measurement infrastructure for DL auto-tuners that dynamically changes parallelism levels to perform fast and accurate latency measurements.
We demonstrate that DOPpler reduces auto-tuning time by 50.5% on average across a large variety of tensor programs, auto-tuners and DL models on both single and multiple-device measurements, whilst maintaining optimization quality equivalent to the default auto-tuning measurement infrastructure.

2 Background

2.1 Deep Learning Optimization

Deep Learning: DL models provide computational abilities to perform tasks such as automatic translation or image recognition. DL models are composed of multi-layered Deep Neural Networks (DNNs) expressed as computational graphs, where nodes represent *operators* (ReLU, Convolution) and edges their data dependencies (N-dimensional tensors) [10]. Operators are implemented by *tensor programs* that comprise CPU code for data loading and accelerator code (e.g. Nvidia CUDA kernels for GPUs) that manipulate and transform tensors. Characterized by their operator quantity, dimensionality and tensor-program complexity, carefully designed DNNs exhibit improved accuracy at the expense of higher Floating Point Operations (FLOP) count [10]. Increasingly, DNN inference is performed on massively-parallel GPU architectures, enabling faster tensor program computation compared to CPUs.

DL inference optimization: DL inference can be performed using DL framework-specific tensor program implementations [11] or DL compilers such as TVM [2]. DL compilers compile high-level DL model definitions to target-device binaries, enabling control over their implementation specifics. Both DL frameworks and compilers can optimize DL tensor programs to reduce their execution latency on a given target-device [12]. At a *high-level*, graph transformations are performed (e.g. algebraic simplification or operator fusion to reduce intermediate program launch steps). At a *low-level*, target-device dependent optimizations are performed (e.g. computation/data alignment to device cache size, vectorization, device-intrinsic function substitution, insertion of synchronization primitives, data tiling towards device threads) [2]. Manually applying low-level optimizations is time-consuming, and must be performed for each unique tensor program towards a unique device, exploring a limited range of the implementation space, which leads to sub-optimal program latency improvement [4].

2.2 Deep Learning Auto-tuning

DL *auto-tuning* enables automation of low-level tensor program optimizations, requiring DL compilers to generate, compile and measure execution latencies of thousands of candidate tensor program implementations on a target-device. The purpose of an auto-tuner is to decide what transformations of *compute*[1] should be performed to parameterize or generate an optimal *schedule*[2], transforming the tensor program implementation. Once measured, an auto-tuner selects the fastest *candidate implementation* for a tensor program. Auto-tuners are similar in operation as shown in Figure 1, however, can be categorized into two types.

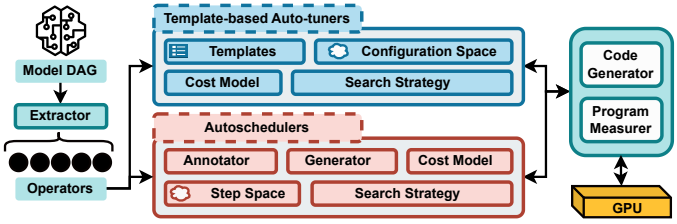


Fig. 1: Overview of DL model auto-tuner operation.

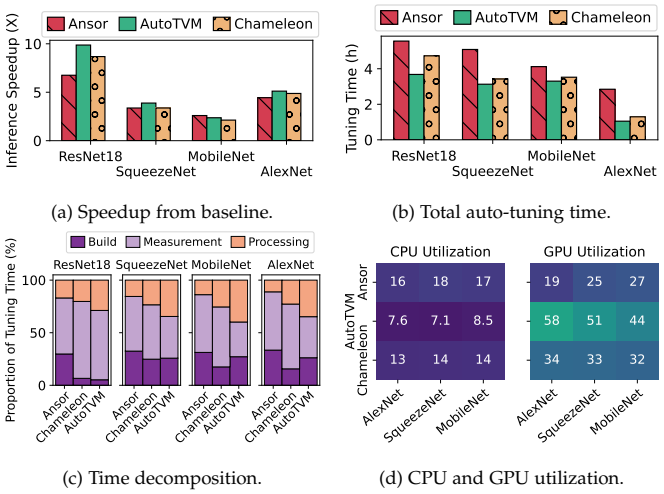
Template-based auto-tuners utilize pre-defined templates that specify an implementation configuration space (e.g. loop tiling or unroll factors). Cost models and search algorithms are used to navigate the configuration space to prune and find candidates that produce lower tensor program latency. Examples include AutoTVM [12], Chameleon [5] or AdaTune [13].

Auto-schedulers generate implementations using rules and steps, incrementally creating a candidate implementation. Search methods and cost-models are used to explore step spaces whilst generating implementations. Exemplified by Ansor [4] or Halide [3], auto-scheduling reduces engineering effort by avoiding manual templates, yet can be less effective for esoteric operators whose implementation spaces are challenging to define within rule-space bounds.

2.3 Tensor Program Execution in Auto-tuning

GPUs – a prominent type of accelerator for DL and the focus of our work – facilitate the Single Program Multiple Data (SPMD) stream programming model, whereby a *kernel* describes operations to be performed on a single stream datapoint. Nvidia GPUs implement SPMD via the Single Instruction Multiple Threads (SIMT) [14] model, where each thread executes the kernel over a single data point simultaneously across multiple GPU cores. DL auto-tuners generate and execute thousands of candidate tensor program implementations, each comprising both CPU-level code and the GPU CUDA kernel(s) to determine candidates with the lowest latency. The auto-tuner measurement infrastructure generates and compiles a batch of tensor programs from configurations provided by the auto-tuner. Each program in the batch is measured (in-order) by (i) spawning a CPU process, (ii) loading tensor data onto the GPU, (iii) collecting a start timestamp, (iv) repeatedly launching the kernel N times (10s - 100s), (v) collecting a stop timestamp, (vi) obtaining average kernel execution latency (from N sample timestamp differences) and, (vii) terminating the CPU process and reporting measurements. This sequence repeats

1. Compute — High-level definition of operator computation.
2. Schedule — Compute implemented towards a target-device.



(a) Speedup from baseline. (b) Total auto-tuning time.

(c) Time decomposition. (d) CPU and GPU utilization.

Fig. 2: Auto-tuning in Nvidia V100 GPU, 500 measurement.

TABLE 1: Hardware, software, and workload setup.

Platform	Hardware Specification
A	2x (16-core) Intel Xeon 5218 [2.3GHz], 196GB DDR4, Nvidia V100 (Volta) 32GB
B	(12-core) AMD Ryzen 1920X [3.5GHz], 128GB DDR4, Nvidia GTX2080 (Turing): 8GB
C	(6-core) Intel i7-6850K [3.8GHz], 32GB DDR4, Nvidia GTX1080 (Pascal) 8GB

Type	Software / Workload Specification
Compiler	TVM 0.7dev [2]
OS/Driver	Ubuntu 20.04, Nvidia Driver 465.31, CUDA 11.3.1 [15]
Auto-tuner	Chameleon, AutoTVM, Ansor [4], [5], [12]
DL Model	MobileNet-V1, SqueezeNet, VGG-16, AlexNet, ResNet-18 [16], [17], [18], [19], [20] { batch 1, $3\times224\times224$ }
Tensor Program	MatMul (x3), T-Conv1D (x2), Conv1D-NCW (x2), Conv1D-NWC (x3), Conv2D-NCHWc-Int8 (x4), Conv2D-HWCN/NCHW/NHWC (x9), GRP-Conv2D, DepthW-Conv2d (x2), Correlation, Dense-Int8 (x2), Dense, Conv3D-NCDHW/NDHWC (x5), T-Conv3D,

for the next candidate in the batch. Existing measurement infrastructure enables single and multi-device candidate execution for measurement on both host-local and remote target-devices via an RPC-based client-tracker-server model [3]. Thus, the degree of auto-tuning speedup is presently determined by the number of available GPUs.

2.4 Performance Limitations in Auto-tuning

Auto-tuning has been shown to effectively optimize DL tensor programs, resulting in sizable DL model inference speedup as depicted in Figure 2a. However, it requires considerable time to complete, as shown in Figure 2b. For example, auto-tuning VGG16 towards Nvidia Titan X using AutoTVM takes >11 hours [5]. The latency measurement phase is particularly time-intensive, constituting on average 51.6% of total auto-tuning time as shown in Figure 2c due to auto-tuners executing individual candidate tensor programs within an isolated target-device to measure latency accurately [4], [13]. The necessity to perform *serial measurement* (i.e one candidate a time) results in high auto-tuning time. This decision is motivated by GPU kernel scheduling unpredictability if multiple candidates were to execute together on a single device. Serial measurement reduces auto-tuning throughput, platform availability and utilization to as low as 7% CPU and 19% GPU as shown in Figure 2d.

3 Achieving Parallel Auto-tuning

Parallelism increases resource efficiency and throughput in many areas of computing. Intuitively, auto-tuning time could also be reduced by launching multiple candidate tensor programs simultaneously on a single target-device during their measurement. However, the established view held and practiced in the community is to measure tensor programs in isolation and serially to ascertain accurate latency. The proprietary scheduling of concurrent CUDA Stream kernels provides no guarantees on workload latency [15] – established to be problematic for parallel candidate measurements. Existing DL auto-tuners are thus designed to strictly perform serial measurement, isolating the device for both remote (RPC) and local auto-tuning. During conventional serial measurements, parallelism is achieved inter-device, using multiple local or remote target-devices where measurements remain serial. To date, no DL auto-tuner performs intra-device parallel measurement.

Parallel Auto-tuning Implementation: Intra-device, parallel auto-tuning on a single target-device could be achieved by binding N TVM RPC servers (see. Section 2.3) to a single GPU, bypassing per-server isolation constraints at the expense of increased overhead with higher *degrees of parallelism* (DP) – the number of simultaneous candidates executing for measurement. Alternatively, existing infrastructure can be modified to disable isolation constraints and extended to enable simultaneous candidate execution intra-device – our approach chosen to investigate parallel auto-tuning. We modified existing infrastructure to spawn separate processes each with a dedicated CUDA Stream, which we define within this work as *Naive Parallel Measurement* (NPM). We further evaluate the effectiveness of using multiple GPUs in Section 5.2.

3.1 Naive Parallel Measurement (NPM)

We compared serial and NPM auto-tuning by studying (i) time taken to complete 500 candidate measurements, and (ii) lowest latency candidate found by an auto-tuner (optimization quality). We optimized 36 tensor programs with 3 auto-tuners Ansor (AN), AutoTVM (AT), Chameleon (CH) in 3 platforms (A, B, C) as per Table 1, with 7 DP levels $f1$ - $64g$. We also studied the impact of Nvidia Multi-Process Service (MPS) [21] that enables kernel multiplexing, avoiding time-slice scheduling and permitting thread blocks to execute kernels from multiple candidate tensor programs.

Auto-tuning time is consistently reduced in NPM across auto-tuners and tensor programs as shown in Figure 3. At higher DP $f8$ - $64g$, NPM achieves 2 - 3.62 speedup over serial, whereas at lower DP $f2$ - $4g$ achieves lesser improvement of 1.01 - 1.2 . This stems from GPU time-slice scheduling, as CUDA Runtime serializes kernels from different Streams. While this is no issue for serial measurement, NPM is unable to fully exploit CPU parallelism

3. A server is instantiated per GPU to measure a single candidate

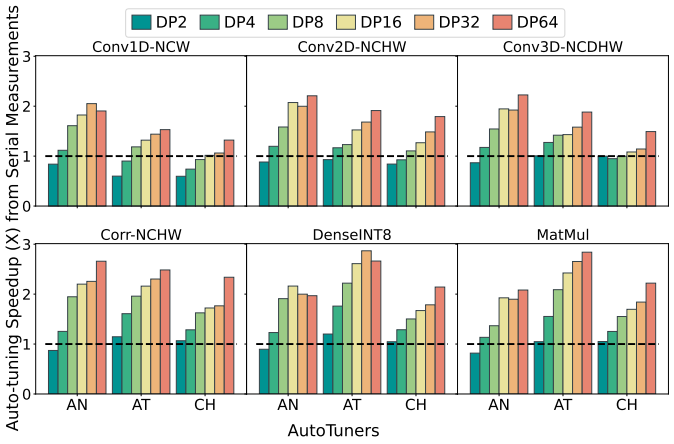


Fig. 3: NPM auto-tuning speedup vs. serial for auto-tuners, tensor programs, and DPs (averaged across platforms).

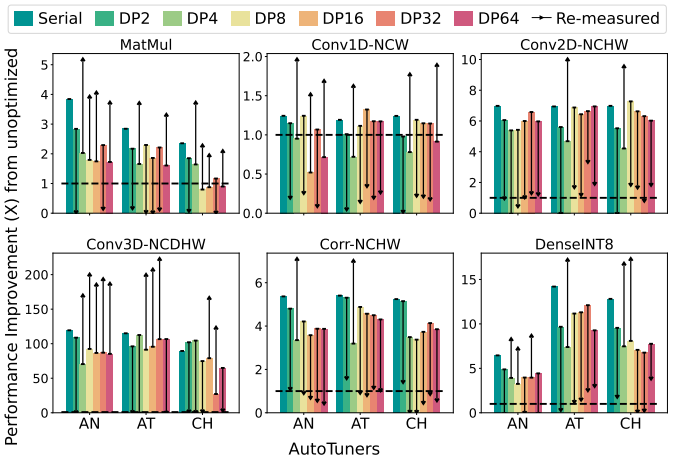


Fig. 4: Performance improvement of NPM auto-tuning vs. Serial. Arrows represent latency achieved by best candidate found during NPM when re-measured serially (in isolation), indicating inaccuracy of NPM measurement at the CPU-level. (averaged across platforms)

due to overhead in process creation and blocking at the CUDA Runtime level, awaiting straggler kernel completion. In few cases, NPM at DP levels $f2$ - $4g$ is slower than serial (Dense, TConv3D-NCDHW). This is due to the default timeout threshold limiting measurement time per candidate to 3 seconds [2], [5] to avoid erroneous candidates preventing auto-tuning progression. In most cases candidate measurement completes within the allotted time, however in the worst case candidates must await timeout before a measurement is deemed failed. For NPM at DP levels $f2$ - $4g$, candidates failing to complete ahead of timeout (e.g. invalid candidate, out-of-memory (OOM) errors) will delay reporting completed candidate measurements until 3 seconds has elapsed, resulting in idle time. At low DP levels this causes total measurement time to be higher than serial measurement. At higher DP levels $f16$ - $64g$ this inefficiency is amortized by the number of in-flight candidates measured. The cumulative measurement throughput from CPU core saturation eliminates idle time between measurements reducing auto-tuning time, however 64 speedup with NPM (DP=64) is rarely possible due to host and target device capacity limitations. Enabling Nvidia MPS achieved auto-tuning speedup between 0.98% to 10.97% (= 25.6%) across all studied platforms and tensor programs.

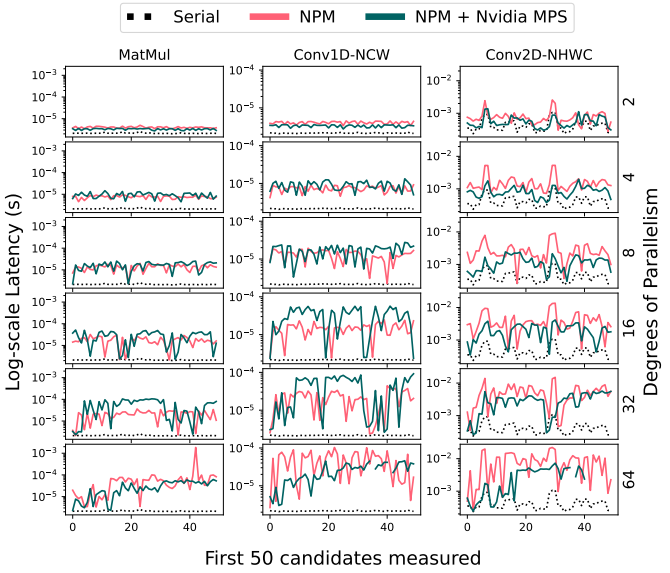


Fig. 5: Candidate latency; Grid-index auto-tuner; Platform A; higher DP produces higher inaccuracy and inconsistency.

3.2 Measurement Inaccuracy

We observed a large discrepancy between measured latency of identical candidates during serial measurement and NPM auto-tuning – which can be expressed as = jr_i r_ij where r_i is the candidate latency i as reported during measurement in isolation, and r_i the latency reported during NPM. As shown in Figure 4, performance improvement achieved with NPM in comparison to baseline tensor-program (i.e. no auto-tuning performed) is typically always worse than serial, however, when the best candidates found by NPM are re-measured in an isolated device (black arrows Figure 4), their reported latency differs substantially from values reported using NPM. When examined with Nvidia Nsight Systems / Compute profilers [22], kernel latencies measured with NPM typically matched latency measurements obtained during serial measurements, suggesting that majority of inaccuracy stems from the CPU-level method of performing measurements. In serial measurement infrastructure, candidate latency measurements rely on *start* and *end* timestamps collected when the kernel launch command is issued via the CUDA Runtime by the tensor program. NPM enables CPU-level parallelism to measure candidate execution. At any DP level 2, kernels originating from separate CUDA Streams (CPU processes) must compete for GPU time, blocking at the GPU scheduler. These factors combined result in unpredictably inflated candidate latency when measured with CPU-level timestamps, and amplified by SM resource constraints and GPU-level scheduling.

3.3 Measurement Inconsistency

The measurement inaccuracy incurred from CPU-level measurements at higher DP levels also hinders the ability of the auto-tuner to converge on low-latency candidates. Figure 5 depicts auto-tuning progress across different tensor pro-

grams when optimizing with a Grid-index auto-tuner [2], which proposes measurement candidates deterministically, allowing for one-to-one latency comparison across different DP levels[4] We observe that for the same candidates, higher DP levels produced greater divergence of the latency distribution compared to serially-measured counterparts, further varying across tensor programs types due to their different computational intensity and scheduling characteristics.

Despite reduced measurement inaccuracy (by 15%, with Median Absolute Deviation 40% across all tensor programs and platforms), introduction of Nvidia MPS still incurs measurement inconsistency, and in some cases further skews the CPU-level timestamp measurements due to kernel funnelling into a single CUDA context. Whilst increasing kernel throughput, MPS exacerbates the overhead of cache/memory accesses by kernels originating from concurrent CUDA Streams, where kernel executions are interleaved and compete for cache/DRAM, particularly for memory-bound tensor programs [15]. Whilst advantageous for some use-cases, we found MPS to be counterproductive for parallel auto-tuning due to negative effects on measurement accuracy. Auto-tuners relying upon inaccurate measurements will explore the implementation spaces of tensor programs differently than serial measurements, and potentially propose sub-optimal candidates.

3.4 Discussion

Our study indicates that when performed in parallel, CPU timestamp-based latency measurements inadvertently incur process blocking (GPU) and competition (CPU) overhead, negatively impacting measurement accuracy. Given this inaccuracy and inconsistency is non-deterministic across measurements, the auto-tuner's ability to effectively explore the implementation space is hindered. As such, a more accurate measurement method is required for NPM to effect a meaningful auto-tuning speedup. Moreover, while naively increasing candidate measurement concurrency reduces auto-tuning time, it leads to more frequent measurement timeouts and failures, reducing the number of successful measurements available for feedback to the auto-tuner. This stems from the static timeout threshold (3s) used in each measurement by default, which when combined with GPU blocking and CPU-level process competition, prematurely preempts measurements that would have succeeded otherwise. These findings suggest that the choice of timeout threshold should reflect a specific DP to minimize candidate failures. Moreover, we observe that certain tensor program types benefit more from NPM w.r.t. time saving and reduced measurement inaccuracy and inconsistency. As such, the relationship between tensor program type and device performance limits must also be considered when deciding appropriate DP that reduces auto-tuning time whilst maintaining accuracy of parallel measurements.

4 DOPpler Design & Implementation

4.1 Overview

The objective of DOPpler is to reduce auto-tuning time and maintain optimization quality. DOPpler achieves this by continually analyzing measurement results, dynamically selecting DP levels to minimize inaccuracy and adjusting measurements reported to the auto-tuner. DOPpler acts as a layer between the auto-tuner and GPU target-device as shown in Figure 6, and comprises: (i) *Precise Parallel Measurer* (PPM), which enables parallel execution of candidates within the same target-device and across multiple devices, whilst increasing parallel candidate measurement accuracy by measuring latency directly on the GPU target-device, as opposed to CPU-level timestamps used by current auto-tuners; (ii) *Calibrator* analyzes measurement consistency, calculates appropriate DP levels for the current measurement batch to adjust reported measurements, and ranked serial re-measurement of a subset of candidates on an isolated target-device to ensure selection of the best candidate. We selected DOPpler's hyperparameters using methods practiced by the TVM auto-tuning community [2], [5], ascertained through prior work [23], [24] and experimenting with different configurations (6 tensor programs, 1 auto-tuner and platform) representing 3.7% of total scenarios evaluated. These hyperparameter values are static across all examined workloads, auto-tuners and platforms, and their effectiveness is studied within a hyperparmeter analysis in Section 5.3. DOPpler is designed towards three goals:

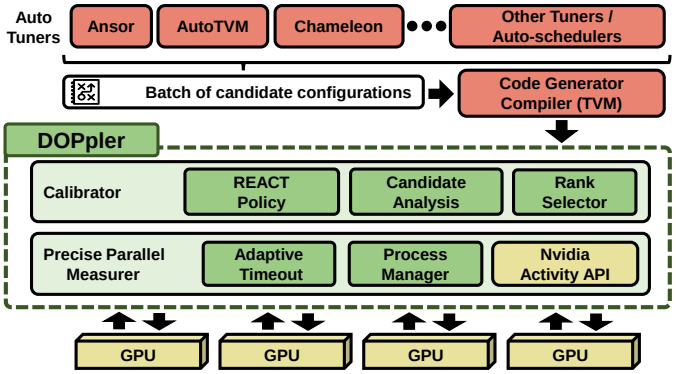


Fig. 6: Overview of DOPpler architecture.

$$\arg\min_{i;j;p} \quad _{mean}; \quad = E(c_i; h_j; d_p) \tag{1}$$

With E denoting the tensor program execution: (i): minimize measurement time $= f_{\ i}\ j\ \ i = 1 : : : Kg$ for a set of candidates $C = fc_i\ j\ i = 1 : : : Kg$ proposed by the auto-tuner given target-device $H = fh_j\ j\ j = 1 : : : Ng$ operating with degree of parallelism $DP = fd_p\ j\ p = 1 : : : max_{dp}g$; (ii): maximize d_p, thus increasing measurement throughput and utilization of the target-device and host CPU, $\arg\max\ d_p$; and (iii): minimize measurement inaccuracy $_{mean}$ resultant from assigned d_p being too high for a given set of candidate measurements executing in parallel.

4.2 Precise Parallel Measurer

The Precise Parallel Measurer (PPM), as shown in Algorithm 1 is a process manager and candidate execution infrastructure leveraging the multiprocessing [25] Python library. The PPM spawns separate CPU worker processes W where

4. Infeasible with auto-tuners such as AutoTVM or Ansor, given their exploration strategies propose different candidates for each new run.

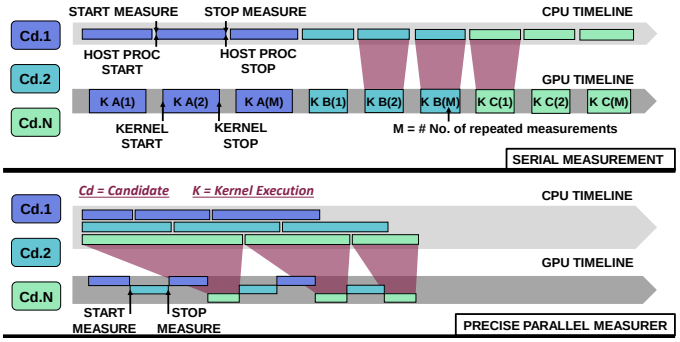


Fig. 7: Serial vs DOPpler Precise Parallel Measurer - same-coloured boxes on each timeline represent repeated execution of the same candidate for averaging (for details, see. 4.3).

$W = \{w_i \mid i = 1:::max_{dp}\}$[5] each containing a separate CUDA Stream, that executes the compiled tensor program b_i and performs latency measurement. Each worker w_i accepts execution requests containing compiled candidates $b_i \in B_{curr}$ and parameters (timeout threshold, number of measurement repeats), via Inter-process Communication (IPC) and/or RPC for multiple local/remote target-devices. The PPM increases measurement throughput at the CPU-level via temporal concurrency, since the CUDA Streams kernels are serialized by the GPU scheduler. DOPpler does not spatially multiplex workloads at the GPU (e.g. by enabling Nvidia MPS) as we found this imposed further measurement inaccuracy with modest measurement time reduction (see. Sections 3.1 and 3.3).

Adaptive timeout: As discussed in Section 3.1, measuring candidates at higher DP levels increases the likelihood of OOM/device launch errors and timeouts. Thus to achieve timely completion of parallel measurements, an adaptive timeout is introduced. The t_{out} denotes the number of seconds until all remaining processes will be preempted due to inactivity or execution latency exceeding t_{out}. We determine an appropriate measurement *timeout* for a DP level using a modified Heaviside step function [23] as follows:

$$t_{out} = b\max\{ ;\min\{ ;2 \quad \tanh(\ d_p\frac{G}{2})gg c \qquad (2)$$

where and represent the minimum and maximum feasible *timeout* bounds, and controls how quickly *timeout* should increase with increasing d_p and G - the set of available target-devices (local or remote). Dynamic *timeout* is directly used by the PPM, as shown in Algorithm 1. We set to conservative 4s (+1 w.r.t. default 3s for serial). We set to 20s, guided empirically by auto-tuning progressively larger tensor programs at $DP = max_{dp}$ [6], where is the maximum time taken for max_{dp} measurements of the largest viable tensor program to complete, whilst avoiding runtime errors.

Measurement accuracy: To alleviate measurement inaccuracy occurring due to CPU-level timestamp collection during parallel measurements described in Section 3.2, we implemented a measurement procedure that collects kernel execution latency information directly from the target-device. We leveraged the Activity API from the Nvidia CUPTI library [26] to collect kernel latencies using the *CUpti_ActivityKernel5* records. Activity API is commonly used by engineers to manually instrument CUDA kernels to profile specific code regions, and is used by the Nvidia profilers [22] to generate detailed profiling reports. We automatized this process by enabling record collection prior to each candidate execution, relying upon the device to report accurate kernel latencies. The difference between serial CPU-level timestamp measurements and DOPpler Precise Parallel Measurer are depicted in Figure 7. Whilst DOPpler does not leverage MPS-based kernel multiplexing and employs accurate kernel latency measurements, any cache/DRAM access overhead resultant from executing tensor program kernels in close succession when submitted in parallel will be inadvertently captured within the Nvidia Activity API on-device measurements. As such, there is an upper bound for DP beyond which measurement inconsistency occurs (see. Section 3.3) resultant from cache/DRAM access competition, and dependent on unique combination of tensor program and target-device hardware.

Algorithm 1: Precise Parallel Measurer

Input: B_{curr}, d_p, G
Output: M_{curr}
init
  W Instantiate max_{dp} workers across G devices
begin
  t_{out} = CalcTimeout(d_p) // Eq. 2
  while *candidates left to measure* **do**
    B_{sel} take d_p G candidates from B_{curr}
    // Allocate d_p candidates to each device $g_i \in G$
    BG_{alloc} allocate(B_{sel}, d_p, G)
    for $(b_i; g_j) \in BG_{alloc}$ **do**
      subm
        measure(wrk=w_{ij},dev=g_{ij},cd=b_i,t=t_{out})
    end
  end
  M_{curr} Retrieve results from workers W
end

4.3 Calibrator

As discussed in Sections 3.1 and 3.3, certain combinations of DP levels and program types exhibit latency inconsistency. To address this, the Calibrator dynamically adjusts DP levels based on changing measurement characteristics. Algorithm 2 describes the Calibrator operation. Initially, the Calibrator compiles tensor program T, $|C|$ times with configurations C provided by the auto-tuner. It then begins a parallel measurement process using Parallel Executor to measure d_p configurations across G devices with DP level = d_p on each device. Any failed measurements are repeated to determine how many failures occurred due to DP being too high ($N_{err:dp}$) and how many succeeded (N_{succ}).

Anatomy of a measurement: Let M denote a batch of K measurements $M = \{m_i \mid i = 1:::K\}$ of candidates $\{b_i \in B\}$ as configured by $\{c_i \in C\}$. In line with current auto-tuners, each execution of b_i is repeated N times during m_i to obtain mean latency of c_i, as depicted in Figure 7. We

5. max_{dp} is the maximum expected DP, set to auto-tuner candidates batch size - commonly set to 64 in analyzed works
6. using the Grid-index auto-tuner towards platform A

Algorithm 2: Calibrator operation

Input: T - Tensor program definition
C - Batch of candidate configurations for T
G - Available target devices
Output: M - Measurements of each B_i as configured by C_i

```
begin
    B ← Compile(T, C)  // B = Set of compiled candidates
    while len(B) ≠ 0 do
        M_res ← Measure(d_p ← G candidates from B, dp=d_p, devs=G)
        M_res, N_err:dp, N_succ ← RemeasureFailed(M_res)
        M_smpl ← SampleCandidates(M_res, param)  // Eq. 3
        M_remeas ← Measure(M_smpl, dp=1, devs=G)
        M_res, μ_mean, σ ← Analyze(M_res, M_remeas)  // Eq. 4
        M_updt ← Update(M_res, μ_mean)  // Eq. 5
        M ← Concat(M, M_updt)
        // REACT Policy Update
        A ← ∞  // Adaptive Max Degree of Parallelism
        if (μ_mean > ε) or (N_err:dp > (d_p ← ε)) then
            A, d_p ← MultiplicativeDecrease(A, d_p, β)  // Eq. 6
        end
        d_cur ← CalculateAdjustment(A, d_p, d_min, d_max)  // Eq. 7
        d_p ← d_p + d_cur // Binary Increase
    end
end
```

denote measurement duration of candidate b_i as $\mu_i \pm 2\sigma$ and mean achieved latency as $l_i \pm L$.

Outlier detection & remeasurement : To identify candidates whose latency measurements was disproportionally affected by parallel execution and unpredictable on-GPU co-scheduling, we performed outlier detection using the double Median Absolute Deviation (MAD) [27] and modified Z-scores [24], adopting the constant 0.6745 from [24]. We feed the detector with a population of ratios between latencies L and measurement duration R calculated as $R = f\frac{l_i}{\mu_i} | l_i \pm L, \mu_i \pm 2\sigma g$. In an ideal scenario, $N \frac{l_i}{\mu_i}$ for each m_i. However, when DP is set too high for the specific tensor program T computational intensity vs. capabilities of the target-device, we found ratios $r_i \pm R$ disproportionally inflated for candidate measurements that were most affected[7] by paralellism, compared to remaining ratios in population R. A total of Q candidates are selected for re-measurement in isolation, comprising outliers and random samples from the iteration batch, where Q is:

$$Q = max(len(R_{outliers}), \delta \cdot len(M_{success})) \quad (3)$$

where $M_{success}$ denotes all successful measurements in the iteration and δ the re-measurement percentage factor, and set to 0.2 based on an initial empirical and sensitivity analysis. A combination of outliers and/or randomly sampled candidates are then re-measured in isolation serially, producing M_{remeas}. Re-measurement results are used to calculate a population-level μ_{mean} as follows:

7. $|l_i - l_j|$ was higher than for other population members, where l_j is the latency of the candidate re-measured in isolation.

$$\mu_{mean} \leftarrow mean\left(\left\{\frac{jl_j \quad l_ij}{l_i} \; j \; l_j \pm L_{remeas}; l_i \pm L\right\}\right) \quad (4)$$

The μ_{mean} is used two-fold: (1) to guide adjustments in DP for next iteration, and (2) to scale measurements reported to the auto-tuner in the current iteration. The scaling is performed as follows:

$$M_{updt} = fr_i \quad (\mu_i \quad (\mu_i \quad \mu_{mean}))jr_i \pm R; \mu_i \pm \sigma g \quad (5)$$

Since scaling is performed using unique per-candidate ratios, each measurement can be individually adjusted w.r.t. population, accounting for individual interference impact.

REACT Policy – DP adjustment: Before commencing a new measurement iteration, Calibrator adjusts DP using the REACT policy – inspired by the operation of the Binary Increase Congestion control (BIC) algorithm [28] from Transmission Control Protocol (TCP). REACT rapidly reacts to any relative change in μ_{mean} and candidate failures $N_{err:dop}$, and attempts to maximize DP as specified in Eq. 1. As per Algorithm 2, during DP adjustment, REACT uses an adaptive maximum d_p for the next round, denoted as A (initialized by). Initially, REACT checks if either μ_{mean} or $N_{err:dop}$ have surpassed threshold , indicating high degree of measurement inaccuracy or failed candidates in the prior round due to the previously proposed d_p. Under such conditions, REACT performs *Multiplicative Decrease* of A using and adjusts d_p as follows:

$$A \quad \begin{cases} jd_p \quad \frac{2}{2}j; & \text{if } d_p < A \\ jd_pj & \text{otherwise} \end{cases} \quad (6)$$
$$d_p \quad jd_p \quad (1 \quad)j$$

Xu et al. (2004) [28] set to 0.125 motivated by higher utilization at the expense of convergence. More conservatively, we set = 0.2, reducing inaccuracy at the expense of throughput. The d_p for next iteration is established using the binary increase factor , as follows:

$$\begin{cases} \frac{A \quad d_p}{2}; & \text{if } d_p < A \\ d_p \quad A; & \text{otherwise} \end{cases} \quad (7)$$
$$maxfminf \; ; \; d_{max}g; \; d_{min}g$$

Where d_{min} and d_{max} are the lower and upper bounds of adjustment at each update point, limiting the degree of change in d_p. In REACT we set d_{min} to 2 and d_{max} to 12 in line with [28]. Calibrator continues to monitor and adjust d_p until auto-tuning completion. Via such approach, DOPpler is able to constantly adjust parallelism levels in response to measurement inaccuracy and target-device compute capability. For example, auto-tuning large tensor programs resulting in high target-device utilization will result in lower DP levels (possibly $d_p = 1$ for very high utilization).

Rank selection: Upon tuning completion, Calibrator re-measures Top-K candidates in isolation to re-affirm the globally best candidate latency. During rank selection, DOPpler reverts to conventional serial measurement. K was empirically determined to 1% of jCj as described in Section 5.3.

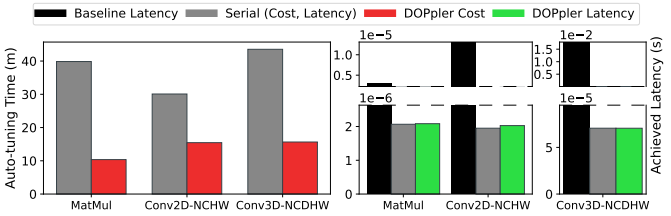


Fig. 8: Auto-tuning 3 tensor programs for 2000 trials, (Ansor, Platform A). DOPpler achieves equivalent performance.

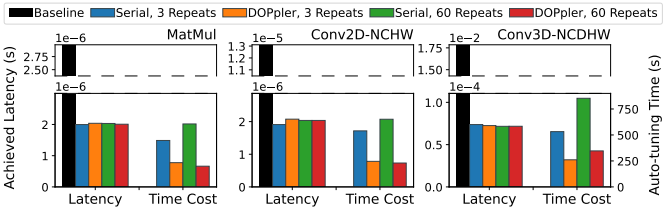


Fig. 9: Impact of repeated measurements with default (3) and DOPpler (60). (Ansor, Platform A). Lowering repeats as a naive speedup method.

TABLE 2: Aggregate results across platforms and auto-tuners. (*) - Auto-tuning with Ansor towards Platform A only

Program Type	DOPpler tuning time reduced %	Serial latency reduced %	DOPpler latency reduced %
Conv1D-NCW	56:24 ± *12.96*	31:62 ± *8.55*	33:18 ± *9.26*
Conv1D-NWC	57:89 ± *13.41*	30:61 ± *8.23*	32:55 ± *7.88*
Conv2D-HWCN	36:41 ± *26.21*	48:57 ± *32.95*	46:19 ± *30.84*
Conv2D-INT8	55:61 ± *13.59*	86:35 ± *3.32*	82:81 ± *9.3*
Conv2D-NCHW	49:28 ± *32.03*	77:74 ± *8.05*	74:81 ± *9.84*
Conv2D-NHWC	49:93 ± *18.31*	63:14 ± *23.04*	66:09 ± *18.99*
Conv3D-NCDHW	47:48 ± *20.48*	98:61 ± *1.54*	97:36 ± *5.12*
Conv3D-NDHWC	48:97 ± *13.08*	94:42 ± *2.36*	94:51 ± *2.3*
Corr-NCHW	55:17 ± *10.04*	80:51 ± *6.59*	81:13 ± *6.28*
DEPTH-Conv2D	45:61 ± *16.31*	30:91 ± *15.78*	25:38 ± *10.53*
Dense	51:47 ± *13.95*	9:56 ± *4.38*	11:49 ± *4.22*
DenseINT8	51:68 ± *15.36*	93:28 ± *3.85*	92:74 ± *4.17*
GRP-Conv2D	52:15 ± *13.91*	97:56 ± *1.31*	98:05 ± *1.01*
MatMul	53:84 ± *21.57*	45:71 ± *9.37*	45:47 ± *8.85*
TConv1D-NCW	57:41 ± *11.85*	54:13 ± *6.23*	56:47 ± *5.28*
TConv3D-NCDHW	46:15 ± *14.65*	88:79 ± *2.52*	88:79 ± *2.77*
*LSTM	65:72 ± *2.56*	36:31 ± *0.90*	36:85 ± *1.70*
*SelfAttention	61:94 ± *8.70*	21:44 ± *3.79*	19:51 ± *1.29*

5 Evaluation

5.1 Experiment Setup

Our goal was to evaluate DOPpler's ability to reduce auto-tuning time while achieving optimization equivalent to that attained using serial measurement infrastructure.

Testbeds: We evaluate DOPler across three platforms listed in Table 1, with primary focus on experiments with on platform A, containing the most computationally powerful target-device at our disposal (Nvidia V100). Each platform contains four GPUs each and is provisioned with identical OS, GPU drivers, TVM compiler version and CUDA compute library as per Table 1.

Auto-tuners: DOPpler was integrated with Ansor auto-scheduler [4] and two template-based auto-tuners; AutoTVM and Chameleon [2], [5] configured with default TVM and auto-tuner parameters (3s timeout, 64 candidate batch size), and settings reported in publications and code-bases [2], [5]. We selected 60 measurement repeats to balance between 3 repeats used in AutoTVM by default, and the TVM's community recommendations of 1000ms measurement duration. Avoiding dynamic number of repeats enables us to compare the measurement infrastructures fairly and proportionally to each tensor program. We evaluate this choice in Section 5.2 and depict results in Figure 9.

Workloads: In evaluation, we used 36 tensor programs (*f*Regular/Transposed*g* *f*1D,2D,3D,Depthwise,Grouped*g* Convolution, Matrix Multiplication etc.) with batch size = 1, and 5 DL Models (AlexNet, SqueezeNet, MobileNet, VGG16, ConvNext [29]) with inputs *f*1 3 224 224*g*. Workloads were chosen in line with existing auto-tuners [2], [4], [5], and experiments repeated 10 times.

Experiments: We performed auto-tuning using current serial measurement infrastructure (*serial*) and DOPpler (*DOPpler auto-tuning*). During, tensor program experiments, auto-tuners complete 500 candidate measurements trials (or early-stop *f*implementation space size < 500*g*). We selected 500 measurements as the stop threshold in line with existing works [5], [13] that perform between 150 and 800 measurements, and commonly observe optimization convergence at 500 trials across majority of experiments. We also demonstrate DOPpler's effectiveness for longer auto-tuning using experiments with 2000 measurements. For DL model experiments we allow auto-tuners to extract all tunable operators, and either perform 500 candidate measurements or self-allocate the number of candidates per operator (Ansor) up to a maximum of *O* 500 candidate measurements, where *O* = number of extracted operators. For multi-device auto-tuning, DOPpler measured *DP* *G* candidates per round where *G* is the number of devices.

Metrics: Metrics collected: (i) auto-tuning time of tensor programs and DL models; (ii) latency of tensor program execution and DL model inference, compiled with best implementation found by the auto-tuner; and (iii) platform utilization of host CPU and GPU target-device.

5.2 Experiment Results

Impact of trials & measurement repeats: Figure 8 compares DOPpler effectiveness during prolonged auto-tuning (2000 trials). We observed that DOPpler produced performance improvement equivalent to serial measurement while reducing total auto-tuning time by 46.5-64.0%. When auto-tuning with repeated measurements reduced from 60 to 3 (see 5.1) DOPpler auto-tuning time marginally reduces as depicted in Figure 9. Compared to serial with 3 repeats, DOPpler reduced auto-tuning time by 47.8–54.4%. This is because the majority of time during serial candidate measurement was spent on process/context management and sequential kernel invocation. We found that in some cases (MatMul) using 3 repeats instead of 60 was counterproductive for DOPpler as it incurred measurement inaccuracy from more frequent candidate re-measurement.

Single-device auto-tuning time: DOPpler was able to attain tensor program auto-tuning times 51.9% 18.6% lower than serial as depicted in Table 2 and Figure 10 (details in supplementary material). This stems from increased measurement throughput achieved by DOPpler, particularly for less-complex or lower FLOP operators (Conv1D-NWC/NCW, TConv1D-NCW, Dense), where auto-tuning completed between 51.4-57.9% faster than serial. Within a minority of experiment runs, DOPpler achieved minimal

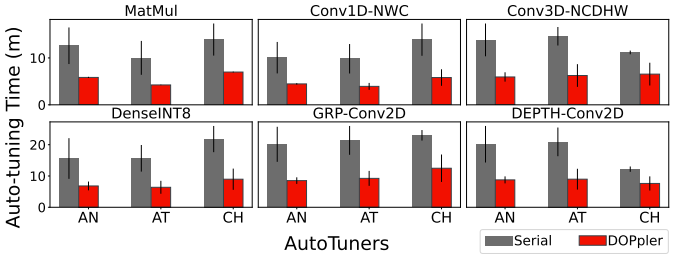


Fig. 10: Auto-tuning time: DOPpler vs. Serial. (Platform A)

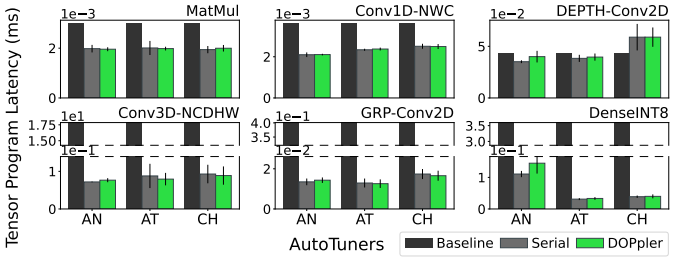


Fig. 11: Tensor program latency: DOPpler vs. Serial. (Platform A)

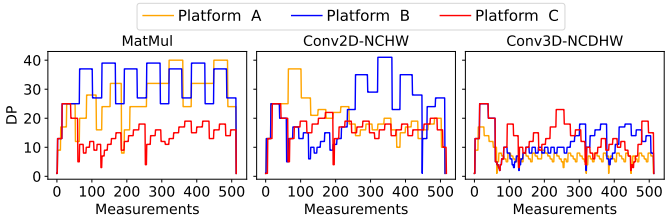
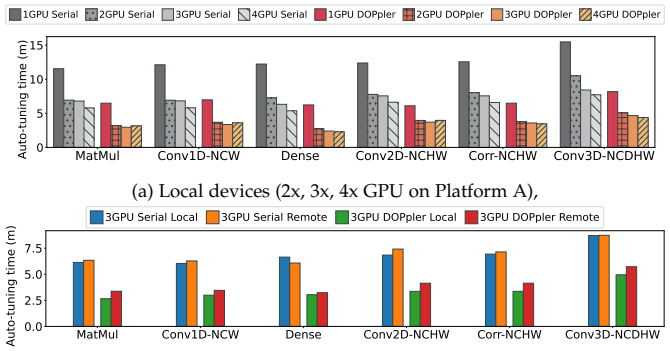


Fig. 12: DOPpler parallelism levels over time (Ansor). Smaller DP levels assigned for large tensor programs.



(a) Local devices (2x, 3x, 4x GPU on Platform A),

(b) Local and Remote devices (3 x Platform B - 1 GPU each)

Fig. 13: Multi-device auto-tuning time (with Ansor)

auto-tuning speedup for certain combinations of tensor program, auto-tuner, and platform (2.53% MatMul, CH, Platform B). Such minimal speed-up was due to a disproportionate number of timeouts or erroneous candidates proposed by the auto-tuner, causing delays. We observed erroneous candidate proposals in Chameleon and AutoTVM (high-FLOP Conv3D-NCDHW). DOPpler increased CPU utilization on platforms A (48.43%), B (54.23%), C (51.46%).

Single-device tensor program latency: DOPpler auto-tuning attained tensor program latency improvement equivalent to serial measurement, reflected by a 1.37% difference on average across all tensor program types as shown in Figure 11 and Table 2. We observed few instances where a measurement method outperformed (Serial: Dense, MatMul; DOPpler: Conv1D-NCW, DEPTH-Conv2D) that is due to non-determinism of auto-tuner exploration strategies. We confirmed this behaviour by auto-tuning a tensor program 10 times using the same auto-tuner, yielding on average 2.16% deviation in achieved latency from variation in implementation space exploration. Thus, the best found candidate latency discovered using DOPpler reside within an equivalent range as serial measurement. For certain combinations (Ansor Dense-INT8, AutoTVM Conv2D-INT8), DOPpler auto-tuning discovered high latency candidates, as esoteric INT-8 tensor programs are known to to consistently generate non-erroneous, high-latency implementations [4].

Parallelism levels: DOPpler assigned varying parallelism levels for different tensor program and platform combinations as shown in Figure 12. Observably, DOPpler reduced DP levels for increasingly larger tensor programs (Conv3D NCDHW, 0.524 GFLOPs) due to increased measurement inaccuracy stemming from kernel interference and contention for compute resources. DOPpler naturally selects and maintains lower DP on less capable devices (Platform C), also adjusting to tensor program complexity.

Multi-device & multi-platform auto-tuning: As shown in Figure 13a, DOPpler auto-tuning on a single GPU yielded a time reduction across multiple local GPUs, providing further time reduction of 52.76%, 52.93%, 45.32% over serial measurement with 2, 3 and 4 GPUs, respectively. For specific tensor programs (MatMul, Conv1D-NCW, Conv2D-NCHW), we observed that using 3 or more GPUs yielded 4.69% improvement compared to 2 GPUs for both DOPpler and serial measurement. This was caused by CPU saturation by candidates on a local device. Such saturation is alleviated when performing auto-tuning across multiple platforms (43.44% vs. 35.06% CPU utilization for local and remote, respectively) providing on average 40.5% time reduction against remote serial measurement as shown in Figure13b. Multi-device and multi-platform DOPpler auto-tuning maintains equivalent optimization quality to serial measurement, with latency differences of 0.97% - 1.46% inline with expected auto-tuner deviation as per Table 2.

Model-level auto-tuning: Following similar trends observed in tensor programs, DOPler auto-tuning also benefited DL models. As shown in Figure 16, DOPpler auto-tuning with Ansor reduced DL model auto-tuning time by 51.40% on average across all models and platforms. We observed that in-line with results shown in Figure 11, DOPpler produced DL model-level inference latency improvement equivalent to those found via serial measurement (= 3:3% 7:8%). Such improvements can also be observed on a per-layer basis as shown in Figure 14 for ConvNext [29](350m parameters, 28 unique tunable layers). DOPpler achieved sizable auto-tuning time reduction (=48.67%, =27.0%) with similar performance gains to serial across majority of layers of varied FLOP count (4:7 e^{6} to 0.92 GFLOP). We observed cases (L5, L11, L17) where DOPpler resulted in slower auto-tuning, stemming from increased number of erroneous candidates proposed by Ansor and thus more frequent re-measurements in DOPpler.

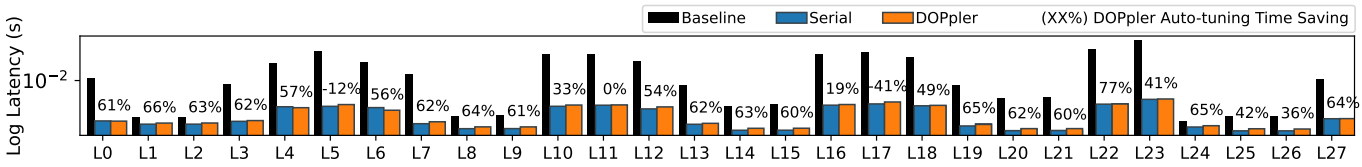


Fig. 14: Performance improvement & auto-tuning time reduction % across ConvNext layers [29]. (Platform A)

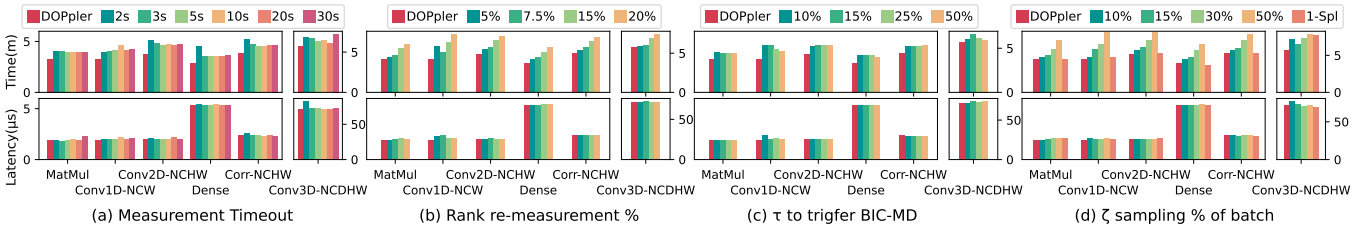


Fig. 15: Hyperparameter analysis. In DOPpler's case, fixed default parameters were used as discussed in Section. 5.3. (Platform A)

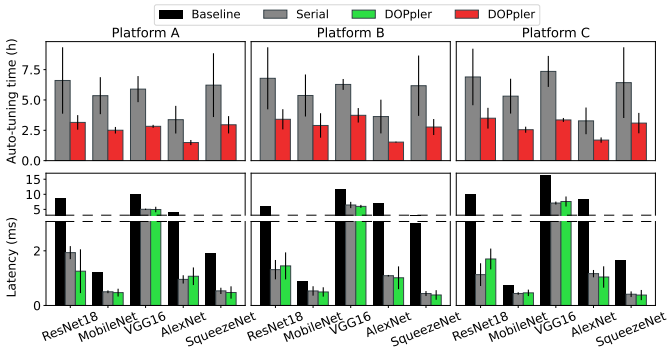


Fig. 16: DL Model-level auto-tuning: DOPpler vs. Serial. (Ansor)

5.3 Hyperparameter Analysis

To demonstrate the effectiveness of DOPpler hyperparameter configuration, we experimentally analysed DOPpler configured with different hyperparameter values.

Dynamic timeout: As shown in Figure 15a, DOPpler's dynamic timeout resulted in 18.84% reduced auto-tuning time over a static timeout value (3s by default in AutoTVM) due to its ability to either avoid excessive OOM, and idle candidate time from configuring lower and higher timeout thresholds based on tensor program type. Such results indicate that there exists no one-size-fits-all timeout threshold applicable to all tensor programs and target-device combinations that can be alleviated by adjusting timeout.

Rank re-measurement: The proportion of re-measured ranked candidates had a sizable impact on auto-tuning time, with a 31.34% difference between 1% (DOPpler) and 20% ranking as shown in Figure 15b. Our intuition was higher re-measurement proportion would find better performing candidates. However, given we observed no major changes to tensor program latency, DOPpler appears to successfully minimize measurement inconsistency, thus rank re-measurement can be configured with a lower value.

threshold: As per Figure 15c, when compared to DOPpler's threshold (5%), we found leveraging a strict (10%) resulted in REACT performing Multiplicative Decrease too frequently, reducing throughput and increasing auto-tuning time. In contrast, using a permissive (25 - 50%) resulted in an excessive DP due to continual Multiplicative Increase in REACT, resulting in increased auto-tuning time due to a higher likelihood of candidate timeout threshold violation and OOM errors discussed in Section 3.

sampling threshold: We compared DOPpler's default 20% against other configurations as shown in Figure 15d. While increasing from 1 sample to 50% negatively impacted auto-tuning time by 30.1%, a low threshold resulted in insufficient samples measured. This caused the calculated $_{mean}$ of the candidate population not exhibiting a sufficiently high $delta$, leading to infrequent Multiplicative Decrease and increased timeout/OOM errors. DOPpler's 20% , whilst re-measuring more samples than other thresholds (10%, 15%, 1 sample), enabled the REACT policy to rapidly respond to variance in measurement accuracy.

5.4 Auto-tuning Large Tensor Programs

We studied DOPpler's ability for auto-tuning large tensor programs that produce high GPU utilization. While large DL models frequently contain many small/medium sized layers, certain models contain tensor programs whose high FLOP-count kernels may lead to higher measurement inaccuracy if executed concurrently. DOPpler was able to successfully auto-tune high FLOP-count tensor programs 0.924–3.70 GFLOPs (larger GFLOP tensor programs produced OOM errors due to NV100 memory capacity) producing high GPU utilization, determined by observing increased instantaneous GPU utilization and more frequent continuous on-device activity compared to serial shown in Figure 17, indicating better GPU utilization and reduced auto-tuning time. Despite higher GPU utilization and approaching NV100 compute limits (Platform A), DOPpler reduced auto-tuning time by 32.4% by avoiding overheads from the sequential candidate execution launch procedure.

DOPpler maintained reasonable performance improvement comparable to serial for high FLOP-count tensor programs as shown in Figure 18. In response to detection of measurement inaccuracy stemming from resource contention, DOPpler was able to assign lower parallelism levels (DP 1-3) when auto-tuning large tensor programs. This is reflected by DOPpler and serial measurement requiring the

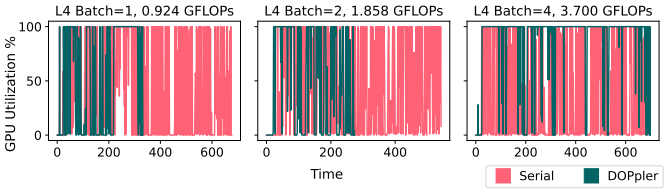


Fig. 17: GPU utilization of auto-tuning Layer 4 ConvNext with varied batch size (1,2,4) sampled every 1s. (Ansor, Platform A)

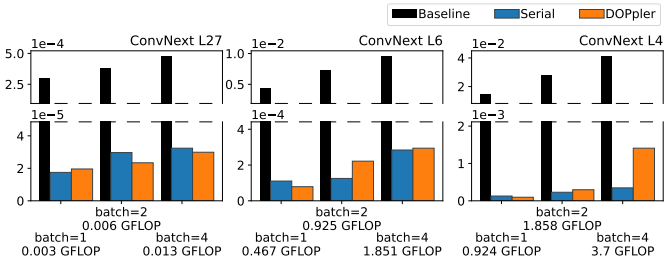


Fig. 18: Auto-tuning performance of large layers. (Ansor, Platform A)

same auto-tuning time for a 3.70 GFLOP tensor program as shown in Figure 17. Whilst reduced DP decreases auto-tuning time reduction, it minimizes likelihood of measurement error from concurrent high GFLOP kernel execution. Effective simultaneous measurement of concurrent high-FLOP kernels is possible due to the time-slice GPU scheduling, which disallows spatially-parallel execution of multiple kernels in the same time-slice [30]. In extreme cases of contention from high-FLOP kernel execution, slowdown occurs due to cache contention which DOPpler attempts to mitigate by reducing DP levels in response to inaccuracy.

5.5 Discussion

Compatibility: DOPpler replaces serial candidate measurement infrastructure currently used by many prominent DL auto-tuners [4], [5] originating from AutoTVM [2]. As such, DOPpler is compatible with all these auto-tuners. DOPpler has been evaluated on Nvidia GPUs due to their prevalence in recent DL research, and especially auto-tuning projects [4], [5]. We believe that DOPpler is also compatible with both edge (Nvidia Jetson - Tegra) and large-scale Multi Instance GPUs such as Nvidia A100, which support Nvidia Activity API and multi-context Stream-based submission of kernels, and have been used to evaluate DL serial measurement auto-tuners [31]. As MIGs' virtual instances entail separate memory, cache and DRAM access paths [32], auto-tuners can target individual instances or the entire GPU, granting DOPpler greater utility given unique instances would require separate auto-tuning. With minimal modification, DOPpler can support non-Nvidia GPUs via OpenCL Cmd-Queues and *clWaitForEvents(...)* [33].

Workload: Whilst evaluated on CNN models and a range of tensor program families, DOPpler is compatible with any DL workload supported by compatible auto-tuners and target-devices. Such compatibility is significant for projects such as Amazon SageMaker Neo [6] that use AutoTVM-derived auto-tuning and Nvidia GPUs, and would benefit from DOPpler's time savings at scale.

Time and quality: Whilst DOPpler speeds-up auto-tuning without compromising optimization, it can balance optimization quality and measurement throughput. Controlling such balance may be advantageous: i.e. during large-scale Neural Architecture Search (NAS) [34], where rapid discovery of DL architectures characterized by high-accuracy and low inference latency is paramount. Alternatively, a DL provider servicing many customers may desire to prioritize throughput over optimization quality (e.g. fast but 'good' auto-tuning immediately, vs. slower and 'excellent' auto-tuning enqueued for hours to days). Such scenarios would require refining or augmenting our proposed DP policy described within Section 4.3.

6 Related Work

DL inference optimization: Improving DL model accuracy has motivated development of various approaches to high-level [35] and low-level [36], [37] inference latency optimizations. Traditionally applied in DL frameworks [11], [38] by leveraging vendor-specific libraries, such optimizations are now bridged and accelerated by DL compilers, enabling more engineering control to obtain high-performance tensor programs for fast DL inference. Several DL-specific compilers have been proposed [39], [40], with prominent types including Halide [41] and TVM [2] that support auto-tuning.

DL auto-tuners: For enabling automatic low-level tensor program optimization, various auto-tuners have been proposed with different search cost models and optimizers to generate fast candidate implementations [13], [41]. Prominent examples include Ansor [4], an auto-scheduler with evolutionary search and gradient-boosting cost model; AutoTVM [2], a template-based auto-tuner with a gradient-boosting cost model and simulated annealing optimizer; and Chameleon [5] using AutoTVM and Reinforcement Learning for candidate proposals and sampling. Whilst parallelizing DL workloads within a GPU has been demonstrated at device-level [42] and cluster-level [43] in other DL systems, auto-tuners have yet to achieve this due to dependency on serial measurements.

Accelerating optimization: Techniques have been proposed to alleviate costly candidate measurements [44], [45] by leveraging trained offline cost models that predict tensor program latency, which requires performing millions of measurements to ascertain candidate latency on unique target-devices to train the model. DOPpler can speedup training and re-training of such predictive optimizers.

7 Conclusion

We present DOPpler, a parallel DL auto-tuning measurement infrastructure. By introducing parallel candidate measurement and addressing multiple auto-tuner performance limitations, we have experimentally demonstrated that DOPpler is capable of achieving significant DL auto-tuning speedup with no degradation to optimization quality. DOPpler grants the capability to perform fast and accurate tensor program optimization for both single and multiple GPU target-devices, which conventionally was thought as infeasible for auto-tuning due to measurement inaccuracy. We hope that our work will aid both researchers and DL service providers towards designing and leveraging DL model optimization tools.

ACKNOWLEDGMENTS

This work was supported by the EPSRC (EP/V007092/1) and the Leverhulme trust Doctoral Scholarships programme in Material Social Futures (DS-2017-036).

REFERENCES

[1] J. Park *et al.*, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *arXiv:1811.09886*, 2018.
[2] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.
[3] A. Adams *et al.*, "Learning to optimize halide with tree search and random programs," *TOG*, vol. 38, no. 4, pp. 1–12, 2019.
[4] L. Zheng *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.
[5] B. H. Ahn *et al.*, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *ICLR 19*, 2019.
[6] Amazon. (2022) Amazon sagemaker neo. [Online]. Available: https://aws.amazon.com/sagemaker/neo
[7] Microsoft. (2023) Watch for. [Online]. Available: https://www.microsoft.com/en-us/research/project/watch-for
[8] OctoML. (2023) Octoml documentation. [Online]. Available: https://app.octoml.ai/docs/
[9] D. Borowiec *et al.* (2023) Conductor + DOPpler. [Online]. Available: https://github.com/dborowiec10/conductor
[10] I. Goodfellow *et al.*, *Deep learning*. MIT press, 2016.
[11] PyTorch. (2022) Pytorch. Facebook's AI Research lab (FAIR). [Online]. Available: https://pytorch.org/
[12] T. Chen *et al.*, "Learning to optimize tensor programs," in *NIPS 2018: The 32nd Annual Conference on Neural Information Processing Systems*, 2018, pp. 3389–3400.
[13] M. Li *et al.*, "Adatune: Adaptive tensor program compilation made efficient," *Advances in Neural Information Processing Systems*, 2020.
[14] E. Lindholm *et al.*, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.
[15] S. Rennich. (2022) CUDA Streams and Concurrency. NVIDIA. [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf
[16] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017.
[17] F. N. Iandola *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv:1602.07360*, 2016.
[18] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.
[19] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.
[20] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[21] Nvidia. (2022) Multi-process service - mps. Nvidia. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf
[22] NVIDIA. (2022) NVIDIA nsight systems. NVIDIA. [Online]. Available: https://developer.nvidia.com/nsight-systems
[23] E. J. Berg, *Heaviside's Operational Calculus as applied to Engineering and Physics*. McGraw-Hill, 1936.
[24] B. Iglewicz *et al.*, *How to detect and handle outliers*. Asq Press, 1993, vol. 16.
[25] Python Foundation. (2022) Python multiprocessing — process-based parallelism. Python Software Foundation. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html
[26] Nvidia. (2022) Cupti :: Cupti documentation. Nvidia. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/index.html
[27] C. Leys *et al.*, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," *Journal of experimental social psychology*, vol. 49, pp. 764–766, 2013.
[28] L. Xu *et al.*, "Binary increase congestion control (bic) for fast long-distance networks," in *IEEE INFOCOM 2004.* IEEE, 2004.
[29] Z. Liu *et al.*, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF*, 2022, pp. 11 976–11 986.
[30] G. Gilman *et al.*, "Characterizing concurrency mechanisms for nvidia gpus under deep learning workloads," *Performance Evaluation*, vol. 151, p. 102234, 2021.
[31] W. Sun *et al.*, "Efficient tensor cores support in tvm for low-latency deep learning," in *DATE 21*, 2021, pp. 120–123.
[32] Nvidia. (2022) Multi-Instance GPU. Nvidia. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/
[33] The Khronos Group Inc. (2022) clwaitforevents. [Online]. Available: https://man.opencl.org/clWaitForEvents.html
[34] B. Baker *et al.*, "Accelerating neural architecture search using performance prediction," *arXiv:1705.10823*, 2017.
[35] L. Wang *et al.*, "A unified optimization approach for cnn model inference on integrated gpus," in *ICPP 19*, 2019, pp. 1–10.
[36] G. S. Murthy *et al.*, "Optimal loop unrolling for gpgpu programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2010, pp. 1–11.
[37] N. G. Dickson *et al.*, "Importance of explicit vectorization for cpu and gpu software performance," *Journal of Computational Physics*, vol. 230, no. 13, pp. 5383–5398, 2011.
[38] Apache Software Foundation. (2022) MXNet. [Online]. Available: https://mxnet.apache.org/
[39] S. Cyphers *et al.*, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," *arXiv:1801.08058*, 2018.
[40] R. Baghdadi *et al.*, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *CGO*. IEEE, 2019, pp. 193–205.
[41] J. Ragan-Kelley *et al.*, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, pp. 519–530, 2013.
[42] S. Pai *et al.*, "Improving gpgpu concurrency with elastic kernels," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 407–418, 2013.
[43] G. Yeung *et al.*, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 88–100, 2022.
[44] B. Steiner *et al.*, "Value learning for throughput optimization of deep learning workloads," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 323–334, 2021.
[45] R. Baghdadi *et al.*, "A deep learning based cost model for automatic code optimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 181–193, 2021.