RESONATOR: ML Autotuning-as-a-Service for Edge-to-Cloud Infrastructure

Junguk Cho†, Ahmed, Faraz†, Cao, Lianjie†, Puneet Sharma†, Dominik Stiller†
†Labs,
{junguk.cho, faraz.ahmed, lianjie.cao, puneet.sharma, dominik.stiller}@hpe.com

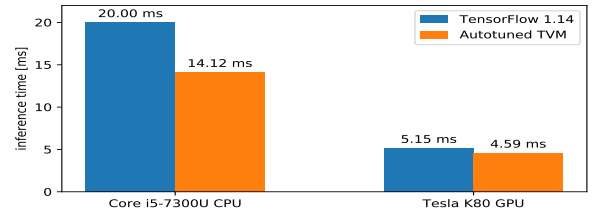
Abstract

Hardware specific tuning of ML models is key to extracting maximum performance from available infrastructure resource. We introduce RESONATOR*, our autotuning service framework to enable optimization of large-scale neural networks without requiring complex configuration by endusers. As one of core components in* RESONATOR*, we design a novel load-aware autotuning scheduler to address existing limitations of autotuning frameworks by interleaving autotuning sub-procedures between multiple autotuning jobs (or even single autotuning job), thus reducing cost of auto-tuning considerably. As a proof-of-concept, we design and implement* RESONATOR *on Kubernetes container management system and showcase the advantage of the proposed load-aware scheduler with multiple commonly used neural network models.*

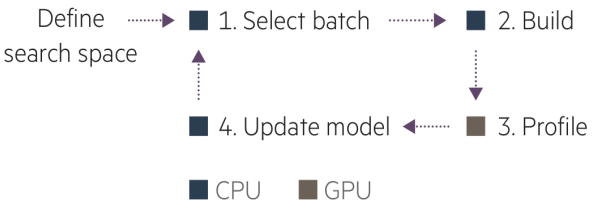
Problem statement

Real-time computer vision applications with deep learning-based inference are being deployed on heterogenous compute hardware including hardware accelerators(e.g., CPU, GPU, FPGA, TPU, etc). Autotuning ML model is to find an optimal low-level implementation of a trained neural network model for specific hardware to improve inference performance. Specifically, it finds optimal loop tiles and ordering, caching, and loop unrolling implementations to reduce memory access cost, maximize parallelism (e.g., CUDA threading), and leverage novel hardware primitives (e.g., Tensor Cores). Performance benefit of autotuning is evident from inference performance of trained MobileNet model from Project Oslo [3]. Figure 1(a) shows 30% and 11% inference performance improvement in autotuned versions against a native tensorflow implementation on CPU and GPU respectively.

Currently, two autotuning frameworks (i.e., TVM [1] and Tensor Comprehensions (TC) [2]) are widely used. Figure 1(b) shows generic autotuning procedures from two frameworks to optimize inference performance on GPU target devices, which consists of four stages: (1) **Select batch stage:** selects a batch of candidates in searching spaces based on cost model stages. In case no initial training data exists, it picks random candidates. (2) **Build stage:** generates executable files based on batch, (3) **Profile stage:** runs executable files and measures execution time on target devices (i.e., GPU), and (4) **Update cost model stage:** uses machine learning (ML) (note that TVM uses a gradient tree boosting model based on XGBoost) to predict executable runtime based on measured execution time. This ML-based cost model is used to guide searching next promising batch in *Select batch stage*. It must be noted that performance improvement comes at cost of deploying autotuning and actual profiling on hardware of interest. It is important to reduce the auto-tuning cost to make it auto-tuning available as a service.



(a) Inference performance comparison.

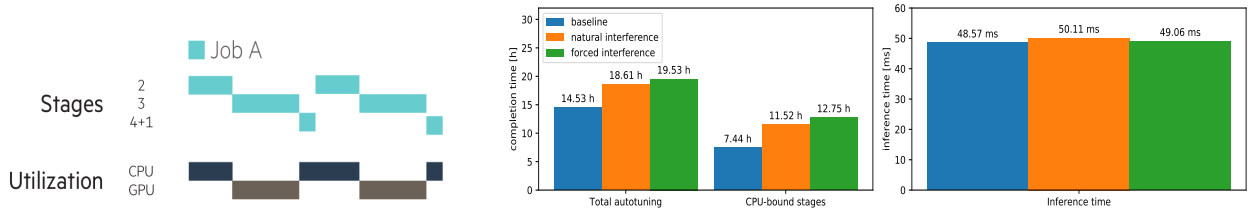


(b) Generic autotuning procedures.

Figure 1: Autotuning

While existing autotuning frameworks show good inference performance, they have fundamental design limitations on autotuning procedures shown in Figure 1(b). This sequential autotuning procedure (i.e., synchronous stages) causes inefficient computation resource utilization (e.g., CPUs, GPUs) to complete autotuning. Figure 2(a) shows a high level resource utilization to run autotuning for GPU devices. Since each stage has strong dependencies, one of computation resources (i.e., CPUs, GPUs) is always idle. For example, while performing *Profile stage* on GPU,

CPU resource is idle since *Update model stage* should wait for measured number from *Profile stage* to update cost model. On the other hand, when performing *Update model stage* on CPU, GPU resource is idle. To avoid this resource inefficiency, running multiple autotuning jobs are a possible solution. However, due to interferences of each stage for multiple autotuning jobs, the completion of autotuing jobs takes longer shown in Figure 2(b). In addition, running multiple autotuning jobs results in higher inference performance since running multiple *Profile stage* on the same GPU resource causes inaccurate profile results. Figure 2(b) shows increase in inference time. Even small differences in inference time make a big difference when a large number of interferences are made (e.g. Seagate, where 1000s of inferences are made everyday)

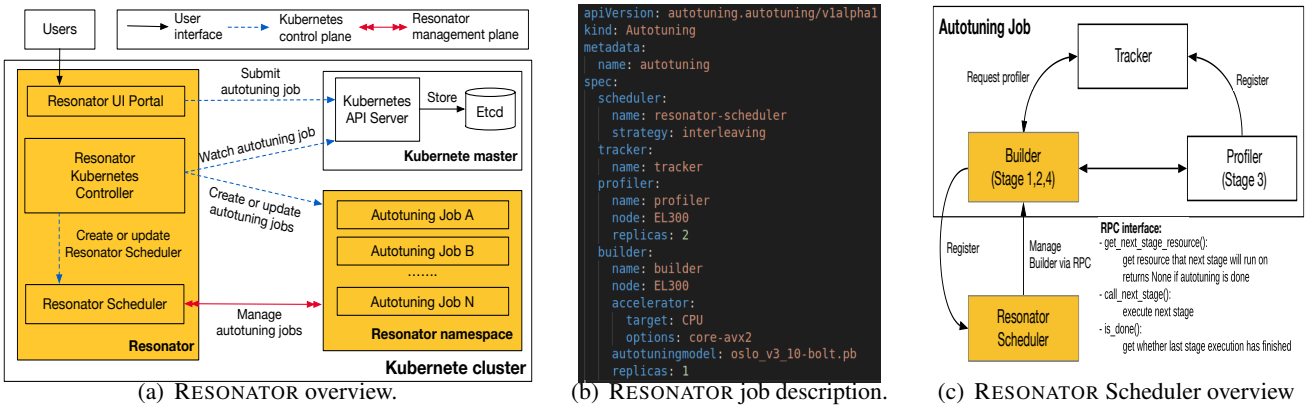


(a) Resource utilization. (b) Effect of interference on autotuning completion and inference time.

Figure 2: Autotuning procedures and resource utilzation.

To address these limitations, we need a solution which (i) maximizes resource utilization and reduces total autotuning completion time, (ii) guarantee optimal inference performance, (iii) provides a simple abstraction to run autotuning jobs by encapsulating the complex hardware and software configurations. (iv) and is pluggable to existing container management systems (e.g., Kubernetes) to start autotuning services without deploying new systems.

Our solution



(a) RESONATOR overview. (b) RESONATOR job description. (c) RESONATOR Scheduler overview

Figure 3: RESONATOR

To meet above requirements, we design RESONATOR- a framework to enable Autotuning as a Service (AaaS). Figure 3(a) shows RESONATOR overview. It consists of RESONATOR UI portal, RESONATOR kubernetes controller and RESONATOR scheduler which run on Kubernetes container management system. To encapsulates complex hardware and autotuning stages in RESONATOR, we define abstractions for resonator scheduler and autotuning stages and realize them as Kubernetes Custom Resource Definition (CRD).

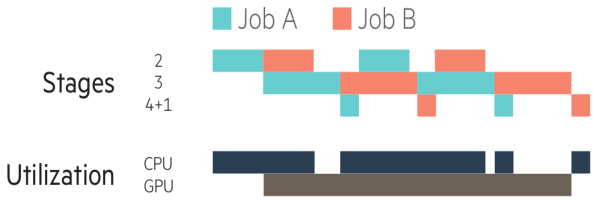
Users for RESONATOR submit their request by only specifying high-level information (e.g., trained model for autotuning, target hardwares to run inference) to RESONATOR UI portal. RESONATOR UI portal converts the request to an autotuning job description and submits to Kubernetes API server. Figure 3(b) shows an example of autotuning job description to optimize a MobileNet model on HPE EL300 Edgeline. RESONATOR Kubernetes controller which watches the kubenetes API server gets notification of a new autotuning job submission. RESONATOR Kubernetes controller starts deploying the new autotuning job on the Kubernetes cluster based on the autotuning job description. In case this is the first autotuning job, RESONATOR Kubernetes controller also launches RESONATOR Scheduler. When the deployment of autotuning job is done, RESONATOR Scheduler manages the autotuning jobs.

Figure 3(c) shows how RESONATOR Scheduler manages deployed multiple autotuning jobs in detail. *Builder*, *Tracker* and *Profiler* components cooperate to complete an autotuning job. *Builder* and *Profiler* perform stage

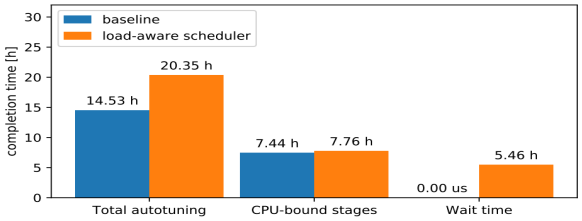
1,2,4 and stage 3 respectively shown in Figure 1(b). *Tracker* is responsible for tracking resource availability (e.g., GPU) in *Profiler*. Before sending executable files to *Profiler*, *Builder* contacts *Tracker* to get available *Profiler* information. RESONATOR Scheduler manages multiple autotuning jobs by controlling each *Builder* which is a starting component for an autotuning job. We design RPC interfaces between RESONATOR Scheduler and *Builder* shown in Figure 3(c). The centralized RESONATOR Scheduler has a global view of multiple autotuning jobs and intelligently interleaves multiple autotuning jobs on different computation resources (i.e., CPUs and GPUs) to run each stage since Figure 4(a) shows a high level view of scheduling each stage for two autotuning jobs on different computation resources. RESONATOR reduces idle time of computation resources and avoids interferences on the shared resources (e.g., CPUs, GPUs).

Evidence the solution works

To validate proposed RESONATOR design, we implement a prototype of RESONATOR shown in Figure 3 and evaluate the prototype in a locally deployed Kubernetes cluster. For autotuning framework, we extend TVM framework to work with RESONATOR Scheduler shown in Figure 3(c). We use ResNet-18 as a benchmark model to optimize inference on Tesla K80 GPUs. We submits two descriptions of autotuning jobs to Kubernetes API server and RESONATOR Kubernetes controller automatically deploys RESONATOR Scheduler and two autotuning jobs on Kubernetes cluster. Figure 4(b) shows autotuning completion time and inference performance. RESONATOR significantly reduces *Builder* operations by avoiding interference from simultaneous *Builder* execution from two jobs. Figure 4(b) shows the same inference time even though we runs two autotuning jobs at the same time by avoiding interference for *Profile stage*.



(a) Ideal resource utilization with RESONATOR.



(b) Results of autotuning with RESONATOR Scheduler.

Figure 4: Results

Competitive approaches

Amazon SageMaker Neo currently provides an autotuning service by extending TVM framework. Since it extends TVM framework, it may inherit the same resource inefficient issues. In addition, since the *Profile stage* should run on the target hardware, if Neo does not support the hardware, it does not provide accurate autotuning results. Instead, our solution enables autotuning services on any Kubernetes clusters on on-prem or public clouds with efficient resource utilization.

Current status and next steps

As described above we have prototyped RESONATOR implementation and evaluated it on various platforms such as HPE Edgeline 300 with different hardware accelerators. Before Techcon 2020, we plan to build a service lifecycle management platform using our current prototype to provide ML autotuning as a service. This will include a hardware workbench with target edge and core compute devices for the profiling stage of auto-tuning. We also plan to extend RESONATOR to leverage hardware accelerators offerings from public cloud service providers. Smart scheduling of RESONATOR to reduce idle duration is even more important in this case given the time-based charging model for accelerators. Once complete RESONATOR can be released as a novel service offering as part of HPE's Bluedata Analytics and AI platform. Evaluation of our current scheduler demonstrate some wait time required for preventing interference. As future work, we are working on techniques to improve scheduler by splitting single job into micro optimization steps for better multiplexing on shared hardware.

References

[1] Tianqi Chen et. al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning " OSDI, 2018.

[2] Facebook, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions", `https://github.com/facebookresearch/TensorComprehensions`

[3] Rajesh Vijayarajan et. al., "Oslo: Machine Learning across multiple sites." HPE Tech Con, 2018.