Towards Neural Architecture-Aware Exploration Of Compiler Optimizations in a Deep Learning {Graph} Compiler

Gaurav Verma
gaurav.verma@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA

Swetang Finviya
sfinviya@cs.stonybrook.edu
Stony Brook University
Stony Brook, NY, USA

Abid M. Malik
amalik@bnl.gov
Brookhaven National Laboratory
Upton, NY, USA

Murali Emani
memani@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Barbara Chapman
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA

ABSTRACT

Deep Neural Networks (DNN) form the basis for many existing and emerging applications. Many DL compilers analyze the computation graphs and apply various optimizations at different stages. These high-level optimizations are applied using compiler passes before feeding the resultant computation graph for low-level and hardware-specific optimizations. With advancements in DNN architectures and backend hardware, the search space of compiler optimizations has grown manifolds. Also, the inclusion of passes without the knowledge of the computation graph leads to increased execution time with a slight influence on the intermediate representation. This paper presents preliminary results 1) summarizing the relevance of pass selection and ordering in a DL compiler, 2) neural architecture-aware selection of optimization passes, and 3) pruning search space for the phase selection problem in a DL compiler. We use TVM as a compiler to demonstrate the experimental results on Nvidia A100 and GeForce RTX 2080 GPUs, establishing the relevance of neural architecture-aware selection of optimization passes for DNNs DL compilers.

Experimental evaluation with seven models categorized into four architecturally different classes demonstrated performance gains for most neural networks. For ResNets, the average throughput increased by 24% and 32% for TensorFlow and PyTorch frameworks, respectively. Additionally, we observed an average 15% decrease in the compilation time for ResNets, 45% for MobileNet, and 54% for SSD-based models without impacting the throughput. BERT models showed a dramatic improvement with a 92% reduction in the compile time.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **General and reference → Performance**.

CF'22, May 17–19, 2022, Torino, Italy

ACM ISBN 978-1-4503-9338-6/22/05…$15.00
https://doi.org/10.1145/3528416.3530251

KEYWORDS

neural architecture, deep learning compilers, pass selection, search space pruning

ACM Reference Format:
Gaurav Verma, Swetang Finviya, Abid M. Malik, Murali Emani, and Barbara Chapman. 2022. Towards Neural Architecture-Aware Exploration Of Compiler Optimizations in a Deep Learning {Graph} Compiler. In *19th ACM International Conference on Computing Frontiers (CF'22), May 17–19, 2022, Torino, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3528416.3530251

1 INTRODUCTION

The burgeoning applications of deep neural networks (DNN) are ubiquitous across multiple artificial intelligence domains, including industry and scientific disciplines. The deep neural architecture has evolved manifolds from simple neural networks to convoluted ones, followed by recurrent ones to massively large models such as Megatron-Turing Natural Language Generation (MT-NLG). Advancements in hardware such as GPU and TPU and DL frameworks like TensorFlow and PyTorch offer optimized kernels support facilitating DL innovations.

The researchers soon identified that the DNN execution differs from the execution of standard computer programs. The network architecture allows extracting parallelism and applying various high-level compiler optimizations specific to tensor operations and particular backend hardware support to these tensor operations. The selection of these compiler optimizations is known as pass selection. The pass selection problem has been researched over decades for traditional computer programs [16, 23–27]. Many of the DL compilers like XLA [34], TVM [5], Glow [33], and TensorRT [31] apply a predefined set of high-level optimization passes on a given input computation graph oblivion to the neural architecture. The optimization search space has exponentially expanded with the increase in custom optimization passes and the evolution of neural network architectures. This explosion in the search space limits the use of static rule-based optimization level selection or the application of machine learning techniques to select the best passes.

In this work, we study diverse DNNs and present the effect of neural architecture-aware selection of passes and execution order resulting in efficient lower-level code generation. We evaluate the experimental results on execution time, throughput, GPU

utilization, memory, and energy consumption metrics. The main contributions of this paper are summarized as follows:

- This work underscores the relevance of neural architecture-aware selection of passes in a DL compiler.
- It evaluates the proposed method against standard optimization level and randomized selection of passes.
- Lastly, we demonstrate how the proposed approach can prune the search space for optimizations selection substantiated with critical metrics.

The remainder of the paper is organized as follows: Section 2 gives the requisite background to understand the problem and presents the related work in this area. Section 3 describes the proposed methodology used in the study. Sections 4 and 5 discuss experimentation and results, respectively. Section 6 provides a conclusion and potential future steps.

2 LITERATURE REVIEW

Pass selection and phase-ordering problems for the compiler writers are decades old but pertinent. With the development of graph-based deep learning compilers, there are manifold possibilities of multilayered optimizations targeting computation graphs generated from an input framework like TensorFlow or PyTorch. After optimizations are applied, the resultant intermediate representation (IR) differs significantly affecting the overall performance. Since optimization passes depend on various factors, including the code block, backend architecture characteristics, and the compiler itself, the search space is enormous, making the selection of passes and ordering an *NP-hard* problem.

In work performed by Haneda et al. [13], the authors propose a statistical technique to reduce the search space for the compiler passes. They evaluated SPECint95 benchmark suite [6] execution on a GCC compiler, validating the heuristics. Similarly, Kulkarni et al. [20] suggest a careful and aggressive pruning of the search space without any information loss. It analyzes the probabilities of various phase interactions, such as inter-phase enabling/disabling relationships and inter-phase independence. Furthermore, research groups [17] have also investigated methodologies involving manually partitioning the optimization phases into independent groups to develop a new multi-stage search algorithm. On average, the performed iterative technique could achieve an 89% reduced search space.

Besides the statistical and iterative heuristics, researchers have also designed machine learning-based heuristics [2, 18] to get to the bottom of efficient execution order. In another work[3], the authors employ clustering-based predictive modeling using dynamic features to attack the problem. They evaluate the results on the Ctuning CBench suite [11] against other earlier discussed heuristics methods. Additionally, a research group implemented a reinforcement learning (RL) based LLVM-derived framework, Autophase [15], to deal with the phase-ordering problem for High-Level Synthesis (HLS) programs.

The effect of phase ordering on energy and power consumption is also investigated for the LLVM based compilers [12, 28]. The experiments exhibit a weak correlation between energy consumption and performance, albeit the authors could significantly decrease the energy consumption and execution time in specific scenarios.

Almost every work discussed so far targets LLVM-based compilers. Currently, DL compilers employ predefined flags, `-O2`, `-O3`, etc. In this work, we show that the search space can be pruned at a higher level, reducing efforts to auto-tune if we make neural-architecture aware selection of passes resulting in reduced execution time and improved throughput. Additionally, it is more relevant to have a compiler-agnostic heuristic involving domain knowledge from neural architecture instead of conventional `-Ox` optimization levels.

3 METHODOLOGY

We started by analyzing the architectural differences in the deep neural networks. Further, we explored various compiler passes available in TVM and their impact on a given neural architecture. Subsequently, we applied them to the deep learning workloads, improving the throughput, latency, and overall performance.

3.1 Neural Architecture Analysis

We considered four different classes of neural networks - ResNet [14], MobileNet [35], Bidirectional Encoder Representations from Transformers (BERT) [10], and Single Shot MultiBox Detector-based (SSD) [22] architectures, as summarized in Table 3. We focused on image classification, object detection, and natural language processing (NLP) task corresponding to Question Answering. Moreover, we assessed the computation graph in TensorFlow, PyTorch, and MXNet to generalize the applicability of our methodology. The precision mode used is FP32 as quantization is not well supported for different networks in TVM. We evaluted a trained network on the same and different dataset to validate the proposition for correctness.

3.1.1 ResNet50: The ResNet is a 50-layer deep convolutional neural network (CNN). To address the accuracy saturation and further degradation problem with increasing depth, it uses a deep residual learning framework. The baseline plain network consists of convolutional layers with a global average pooling layer and a fully connected (FC) layer with a softmax activation function in the end. It passes through phases performing the convolution with stride 2, batch normalization, and ReLU activation followed by the multiplication with the weight matrix. The zeros are padded, matching the dimension when there is an increase in the dimension. We have summarized ResNet50 architecture in Table 1.

Table 1: ResNet50 Architecture Summary [14]

Layer_Type	Output_Size	Building_Blocks
conv1_*	112X112	7X7, 64, stride 2
conv2_*	56X56	3X3 max pool, stride 2
		[1X1, 64, 3X3, 64, 1X1, 256] X 3
conv3_*	28X28	[1X1, 128, 3X3, 128, 1X1, 512] X 4
conv4_*	14X14	[1X1, 256, 3X3, 256, 1X1, 1024] X 6
conv5_*	7X7	[1X1, 512, 3X3, 512, 1X1, 2048] X 3
-	1X1	average pool, 1000-d FC, softmax

3.1.2 MobileNetV2: MobilNetV2 is derived from an inverted residual structure where the residual connections are between the bottleneck layers. The basic building block is bottleneck depth-separable convolutions consisting of residuals. As shown in Table 2, it consists of fully connected layers with 32 filters and 19 residual layers. It

further employs ReLU6 as the activation function, and the kernel size is 3x3.

Table 2: MobileNetV2 Architecture Summary [35]

Input	Operator	Expansion Factor (t)	#Output Channels (c)	Repetition times (n)	Stride (s)
224^2X3	conv2d	-	32	1	2
112^2X32	bottleneck	1	16	1	1
112^2X16	bottleneck	6	24	2	2
56^2X24	bottleneck	6	32	3	2
28^2X32	bottleneck	6	64	4	2
14^2X64	bottleneck	6	96	3	1
14^2X96	bottleneck	6	160	3	2
7^2X160	bottleneck	6	320	1	1
7^2X320	conv2d $1X1$	-	1280	1	1
7^2X1280	avgpool $7X7$	-	-	1	-
$1X1X1280$	conv2d $1X1$	-	k	-	-

3.1.3 SSD_ResNet50: The Single Shot (SS) in SSD refers to the object localization and classification tasks performed in a single forward pass of the network. The SSD network is based on a feed-forward convolutional network consisting of feature maps extraction and object detection using a convolution filter. Its uniqueness is that the final fully connected layers in the original ResNet are replaced by the SSD head, as shown in Figure 1. The SSD head utilizes the spatial information extracted by the ResNet to decide the bounding boxes and predict classes.

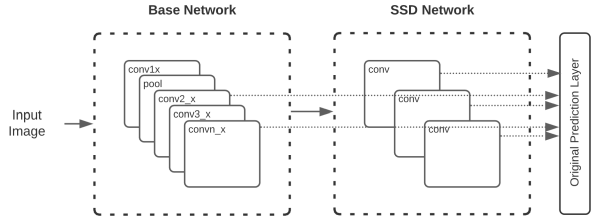


Figure 1: Architecture of a Convolutional Network with SSD Layers

3.1.4 BERT: BERT is essentially a multi-layer bidirectional Transformer encoder based on the attention mechanism [36]. The transformer architecture employs self-attention on the encoder side and attention on the decoder end. It consists of parallel FC layers and transpose operations. Further scaling is performed before passing through the softmax layer to output probabilities. We have used $\text{BERT}_{\text{BASE}}$ extended model in this work. It has 12 layers in the encoder stack, 768 feedforward hidden units, and 12 attention heads.

Table 3: Models' Specifications

Model	Train_Dataset	Test_Dataset	Framework	DL Task
ResNet50	ImageNet	ImageNet, CIFAR10	TensorFlow, PyTorch	Image Classification
MobileNetV2	ImageNet	ImageNet, CIFAR10	PyTorch	Image Classification
SSD_ResNet50	COCO	COCO	PyTorch, MXNet	Object Detection
$\text{BERT}_{\text{base_cased_squad}}$V2	SQuAD_V2	SQuAD_V2	PyTorch	Question Answering

3.2 Compiler Optimizations Analysis

In DL compilers, the passes are categorized into optimization levels (`OPT_LEVEL`), `-Ox`, identical to the conventional compilers. We employed the domain knowledge from the neural architecture analysis to classify the optimization passes. While writing this paper, we listed passes and their functionality available in TVM. The neural network layers, tensor operations, and their order dictated the categorization of passes. It is observed that certain passes are applicable only when a particular feature is supported by a compiler and is available in a neural network. For example, `FoldExplicitPadding` is relevant to a network with explicit padding. Using such a pass for a network like BERT will only increase the search space of the optimizations. We compiled the models under different pass combinations and examined the IR to validate this.

We studied the passes executed as part of `OPT_LEVEL=3` to establish the baseline results. Our observations supported the following two suppositions. Firstly, two or more optimization levels can produce precisely the same IR. For example, `OPT_LEVEL=2` and `OPT_LEVEL=3` generated the same IR for ResNet50. Secondly, an optimization level may contain a set of passes that do not affect the IR. For example, as part of `OPT_LEVEL=2`, `DynamicToStatic` converts dynamic operations to the static, if possible. But all the employed networks have static operations alone.

Based on the above characterization, we selected passes relevant to the experimented neural networks as summarized in Table 4 from all the available passes in TVM. We have classified them into the following broader categories:

- **Baseline Passes (BL):** These are the passes enabled as part of OPT_LEVEL=3. We have used them as our baseline experimentation.
- **ResNet Class (RN):** These passes are relevant to ResNet neural architecture.
- **MobileNet Class (MN):** These passes are explored as part of the MobileNet neural network.
- **SSD Class (SSD):** This class refers to the passes relevant to the SSD network.
- **BERT Class (BR):** These passes align with the BERT architecture.
- **Additional Passes (AD):** There are certain passes, like `ToMixedPrecision` that are dependent on the users' intent. It can be employed across networks.

This classification is intended to increase with the addition of more passes. A particular pass can belong to more than one class. While executing SSD_ResNet, we can combine RN and SSD classes to form the search space. This technique could reduce the search space for the compiler optimizations selection, diminishing the overhead.

4 EXPERIMENTAL SETUP

We evaluate our proposed methodology on GPUs with different architectures. We decided to consider different compute backends based on the previous results presented by Verma et al. [37].

4.1 Hardware Specifications

The experiments are performed on two Nvidia GPUs, GeForce RTX 2080 and A100. GeForce RTX 2080 provides dedicated ray-tracing and CUDA cores. A100 is based on the bests from Volta and Turing architectures. It offers a larger and faster L1 cache and shared memory units. Further, more on-chip memory, including a 40 MB L2 cache, improves the computing performance. Lastly, the CUDA graphs improves efficiency by launching many kernels in a single operation.

Table 4: Categorization of Passes

Pass	Description	Category
AlterOpLayout	Used for computing convolution in custom layouts or other general weight pre-transformation.	BL; RN; MN; SSD; BR
AnnotateSpans	Annotate a program with span information	AD
BatchingOps	Batching parallel operators into one for Conv2D, Dense and BatchMatmul	BR
CanonicalizeCast	Canonicalize cast expressions to make operator fusion more efficient.	BL; RN
CanonicalizeOps	Canonicalize special operators to basic operators	BL
CombineParallelConv2D	Combine multiple conv2d operators into one	RN
CombineParallelDense	Combine multiple dense operators into one	RN
ConvertLayout	Alternate the layouts of operators	BL
DeadCodeElimination	Remove expressions that do not have any usage	RN; MN; SSD
DefuseOps	Inverse operation of fusion pass.	BL
DynamicToStatic	Convert dynamic operations to static if possible	AD
EliminateCommonSubexpr	Eliminate common subexpressions	BL; RN; MN
FakeQuantizationToInteger	Takes fake quantized graphs and convert them to actual integer operations	AD
FastMath	Convert expensive non linear functions to their fast but approximate counterparts	MN; BR; SSD
FirstOrderGradient	Transform all global functions in the module to return the original result and the gradients of the inputs.	MN
FoldConstant	Fold the constant expressions in a Relay program	RN; MN; SSD
FoldExplicitPadding	Find explict padding before an operator that supports implicit padding and fuses them.	RN; SSD
ForwardFoldScaleAxis	Fold the scaling of axis into weights of conv2d/dense	BR
FuseOps	Fuse operators in an expression to a larger operator	RN; MN; SSD; BR
MergeComposite	Merge multiple operators into a single composite relay function	RN; MN; SSD
PartitionGraph	Partition a Relay program into regions that can be executed on different backends	AD
RemoveUnusedFunctions	Remove unused global relay functions in a relay module	SSD
SimplifyExpr	Simplify the Relay expression, including merging consecutive reshapes.	RN; MN; SSD
SimplifyFCTranspose	Simplify the transpose operation on a dense layer	MN; BR
SimplifyInference	Simplify the data-flow graph for inference phase.	RN; MN; SSD
ToANormalForm	Turn Graph Normal Form expression into A Normal Form Expression	AD
ToGraphNormalForm	Turn a normal form into graph normal form.	RN; MN
ToMixedPrecision	Automatic mixed precision rewriter	AD

*BL: baseline optimization passes having OPT_LEVEL=3; RN: ResNet Class; MN: MobileNet Class; SSD: SSD_ResNet Class; BR: BERT Class; AD: Additional optimizations

The tensor core enabled A100 has Thermal Design Power (TDP) of 400 W, whereas CUDA core enabled GeForce 2080 has a TDP of 250 W. The A100's memory bus width is almost 15x of the GeForce. Also, the memory bandwidth of A100 exceeds GeForce by 2.5x. It enables A100 for faster data access and faster data processing.

4.2 Software Specifications

We carried out the experiments with TensorFlow v2.4.0, Torch v1.7.0, Torchvision v0.8.1, Pytorch-transformer v1.2.0, and MxNet v1.8.0 with Python v3.6.9 and TVM v0.8.dev0. To maintain the consistency of the CUDA and CuDNN versions, we utilized CUDA v11.0 and CuDNN version v8.0 for both A100 and GeForce GPUs.

To profile the executions at a system level, we used NVIDIA Nsight Systems version 2021.3.2.4-027534f [30], and for the kernel level profiling, we used NVIDIA Nsight Compute Command-Line Profiler Version 2021.2.2.0 (build 30282580) [29].

4.3 Dataset

In this section, we summarize the dataset's statistical information used to evaluate the proposed methodology.

4.3.1 ImageNet: The ImageNet dataset [9] is a collection of human-annotated images. It offers variations of the same object, including camera angles and lighting conditions. ImageNet has more than 14 million images organized into over 21,000 subcategories averaging around 500 images per subcategory and 21 high-level categories. There are 1000 synonym sets and 1.2 million images with Scale-Invariant Feature Transform (SIFT) features.

4.3.2 Common Objects in Context (COCO): COCO [21] is a large-scale object detection, segmentation, and captioning dataset. It consists of everyday scenes comprising common objects in their natural context. There are 165,482 train, 81,208 validation, and 81,434 test images encompassing 91 categories. The major portion of the dataset is non-iconic images, as they are better at generalizing.

4.3.3 CIFAR-10: CIFAR-10 [19] is a labeled subset of the 80 million tiny images dataset. It consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is separated into five training and one test batch. The test batch contains 1000 randomly-selected images from each class.

4.3.4 Stanford Question Answering Dataset (SQuAD2.0): SQuAD [32] is a reading comprehension dataset comprising questions posed by participants on a set of Wikipedia articles. It consists of 100,000 answerable questions and over 50,000 unanswerable questions. The questions are structurally indistinguishable.

5 RESULTS

We executed each neural network for 100 warm-up runs and 1000 runs to gather the stats to avoid noise. Also, we considered only three standard deviations of the collected data from the mean and excluded any outliers. We selected the following metrics to assess the performance of our proposition.

- **Throughput:** the volume of inferences within a given period, usually measured in inferences per second.
- **Latency:** the execution time to perform inference on one image, expressed in milliseconds (ms).
- **Compile Time:** the time required to generate the optimized computation graph to be deployed; expressed in seconds (sec).
- **Power:** refers to the power drawn by the GPU to perform one inference. It is expressed in Watt (W),
- **Memory Used:** refers to the total memory allocated by active contexts (MiB).
- **Temperature:** refers to the core GPU temperature (°C).

The following notations are used to represent different selections and ordering of passes.

- **BL:** selection of passes having `OPT_LEVEL=3`.
- **AS-0** and **AS-1:** architecture-aware selection of passes for a given class; v0, v1. Table 4 is referred to get the class and varying permutations are considered to get the best result.
- **PO-0**, **PO-1**, **PO-2** and **PO-3:** randomized selection of passes; v0, v1, v2, v3. Class information and additional passes from

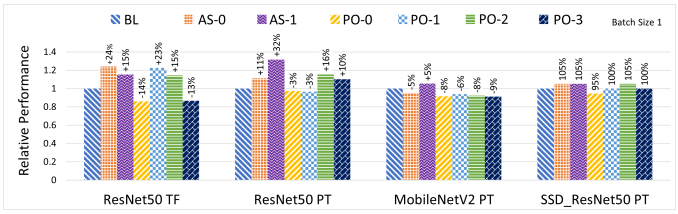
Table 4 are considered to find the random set of passes and then a sequence is proposed based on multiple trials.

Table 5: Pass selection and ordering for SSD_ResNet50 in PyTorch

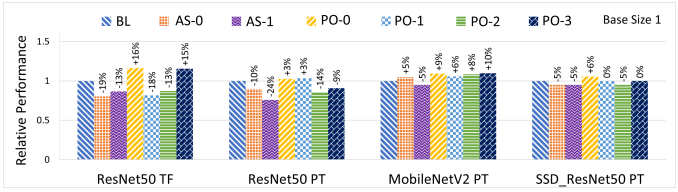
ID	Selected Passes and Ordering
BL	AlterOpLayout, CanonicalizeCast, CanonicalizeOps, ConvertLayout, DefuseOps, EliminateCommonSubexpr
AS-0	AlterOpLayout, FuseOps, SimplifyExpr, FoldConstant, DeadCodeElimination, MergeComposite, FastMath, RemoveUnusedFunctions
AS-1	SimplifyExpr, FuseOps, AlterOpLayout, MergeComposite, FastMath, DeadCodeElimination, FoldConstant, RemoveUnusedFunctions
PO-0	AlterOpLayout, CombineParallelConv2D, DefuseOps, DynamicToStatic, CanonicalizeOps, CanonicalizeCast
PO-1	CanonicalizeCast, AlterOpLayout, DefuseOps, CombineParallelConv2D, PartitionGraph, FakeQuantizationToInteger
PO-2	ToMixedPrecision, CombineParallelConv2D, EliminateCommonSubexpr, SimplifyFCTranspose, CanonicalizeOps, DefuseOps, ToGraphNormalForm, ToGraphNormalForm
PO-3	CombineParallelDense, FakeQuantizationToInteger, AlterOpLayout, CombineParallelConv2D, ToGraphNormalForm, CanonicalizeOps

The pass dependency is handled internally. If a pass depends on the execution of another pass, it is called internally during the execution. In Table 5, we have presented the selected pass and ordering for the SSD_ResNet50 neural network in PyTorch.

5.1 Experiments on GeForce RTX 2080



(a) Variation of "Throughput" with Pass Selection



(b) Variation of "Latency" with Pass Selection

Figure 2: Execution on a GeForce RTX 2080 GPU

We evaluated the performance using seven sets of passes to formulate the pass selections similar to Table 5. As explained earlier, it is based on the neural network's categories presented in Table 4. Where AS-x is the architecture-specific selection, PO-x is the randomized selection of passes from the reduced search space. We further permuted each PO-x version to achieve the best performance among the selected version.

As shown in Figure 2a, the baseline throughput (frames/sec) for ResNet50 TF, ResNet50 PT, MobileNetV2 PT, and SSD_ResNet50 PT are 23.18, 21.35, 72.90, and 3.80, respectively. For ResNet50 implementation in TensorFlow, we achieved up to 24% improvement in the throughput with an informed selection of passes. Similar behavior was observed in the PyTorch implementation of ResNet50. Since ResNet is primarily a convolutional layers followed by the FC layer, we applied default passes like `AlterOpLayout` and `FoldConstant` followed by passes specific to the tensor operations like `FuseOps`, `EliminateCommonSubexpr`, and so on. It is observed that `FuseOps` after `AlterOpLayout` performs better as it leads to more efficient fusion. Hence, we were able to achieve an improvement of 32% in terms of throughput.

On the contrary, throughput for MobileNet and SSD-ResNet did not improve much on a non-tensor core architecture. There was a 5% improvement on average. MobileNet is a lightweight architecture primarily consists of bottleneck layers containing fewer nodes than the previous layer. Hence, this network class does not have many relevant passes. Similarly, for the SSD_ResNet, the addition of new SSD layers reduces the overall gain. Analogous behavior was noticed with the latency gain. As shown in Figure 2b, the baseline latency (ms/inference) for ResNet50 TF, ResNet50 PT, MobileNetV2 PT, and SSD_ResNet50 PT are 43.13, 46.83, 13.71, and 263.16, respectively. The neural architecture-aware selection of optimizations reduced the latency by 18%-19% in the case of ResNet50 in TF, and up to 24% in PyTorch implementation, as shown in Figure 2b. We achieved up to a 5% latency gain for the MobileNet and SSD_ResNet models.

5.2 Experiments on A100

The NVIDIA A100 GPU is a tensor core supported hardware that offers advanced support for tensor operations, mainly CUDA graphs acceleration, as discussed before. As shown in Figure 3, pruning the optimization passes search space and selecting lesser and more relevant passes than `-O3 OPT_LEVEL` improves the compilation time by 18% in ResNet50 TF and by 15% in the PyTorch implementation. Unlike PyTorch, on using combinations of `MergeCompilerRegions` and `SimplifyInference`, the compile-time in TF almost doubled the baseline compile-time. It is due to the aggressive traversal of the computation graph without optimizing the IR in TF. Currently, we are investigating the detailed cause of this behavior. One possible explanation is the difference in the implementation of operators in TF and PyTorch, and hence the difference in the computation graph generated in either case.

The architecture-aware selection of passes was paramount to MobileNet and SSD_ResNet. In both scenarios, we could reduce compile-time by almost 45%-54%. As shown in Table 4 and 5, the fewer optimization passes reduced the overhead of traversing a vast computation graph. While we could cherry-pick passes like `MergeComposite`, and `FastMath`, passes like `DynamicToStatic` and `FakeQuantizationToInteger` could be excluded from the AS-x versions. We validated our results across different frameworks. SSD_MobileNet implementation in MXNet showed similar behavior. On evaluating different classes of models, we found that the new tensor core offers hardware optimizations that diminish the benefits performed by various compiler optimization passes selected as part of `-O3 OPT_LEVEL`. Since A100 comes with tensor cores and advanced CUDA compute capabilities as described in section 4.1, we also evaluated the computation graph without any optimization passes in scenarios where the hardware would do most optimizations. For SSD-based models across formats, it performed on par with baseline execution.

The most exciting results came from the BERT's architecture-aware selection of passes. We could achieve an almost 92% reduction

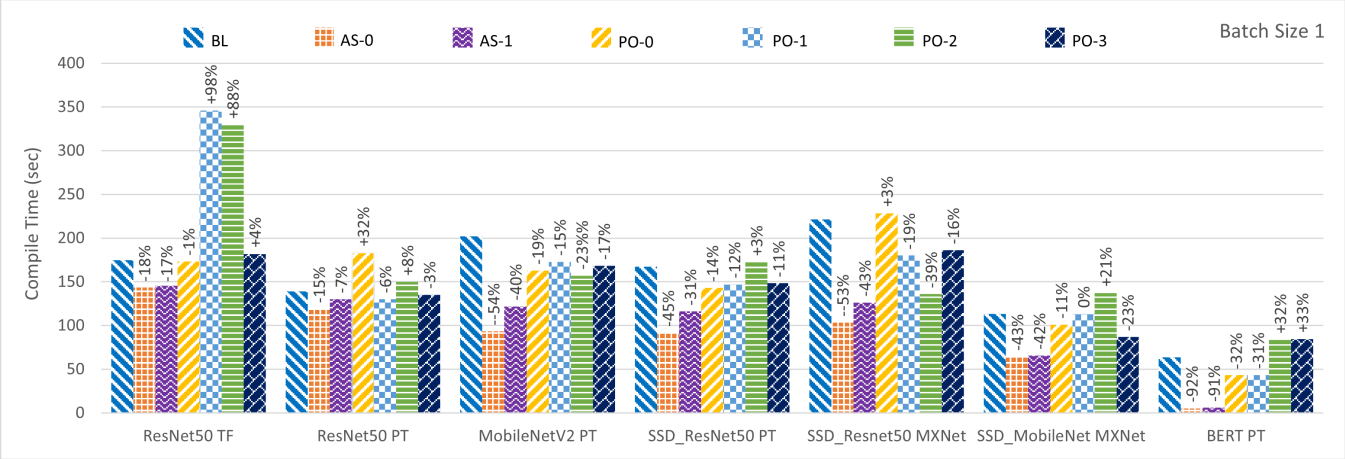


Figure 3: Compile-Time Reduction on A100 GPU

in the compile-time without reducing the throughput. That is critical when we need to Just-in-Time (JIT) compile for the edge devices. We found that only a few passes were relevant due to the BERT-based model's transformer-based architecture. Hence we narrowed down the search space to the BR class passes ad selected passes like `SimplifyFCTranspose`, `FastMath`, `FoldExplicitPadding`. BERT employs three parallel FC layers followed by three parallel transpose operations in a self-attention layer. Also, it performs scaling and softmax that get benefited by including `FastMath`. Furthermore, it uses padding extensively for the shorted inputs. Additionally,

Table 6: Selected Models with Considerable Distinctions

Compute Hardware	Model	Framework	Pass Order	Power (W)		Temperature (°C)		Memory Util. (MiB)	
				Median	Max	Median	Max	Median	Max
GeForce RTX 2080	ResNet50_V2	TensorFlow	-O3	48	50	29	32	601	650
			AS_{best}	**43**	**46**	28	32	**592**	**638**
		PyTorch	-O3	47	49	33	35	456	504
			AS_{best}	46	49	**29**	**32**	**433**	504
	SSD_ResNet50	PyTorch	-O3	90	134	31	37	1287	1458
			AS_{best}	**83**	133	**30**	**34**	**1160**	1450
	MobileNet_V2	PyTorch	-O3	42	45	28	31	174	190
			AS_{best}	**41**	**43**	28	29	**154**	**186**
A100	SSD_ResNet50	MXNet	-O3	80	119	26	30	1747	1887
			AS_{best}	**73**	**110**	**25**	**27**	**1723**	**1879**
	MobileNet_V2	PyTorch	-O3	59	63	23	26	1552	1650
			AS_{best}	58	63	23	25	**1540**	1650
	BERT	PyTorch	-O3	109	249	26	36	1782	2586
			AS_{best}	104	250	26	36	**1680**	2586

*AS_{best}: Neural Architecture-Aware selection of passes.

we gathered hardware statistics, namely, power consumption, GPU temperature, and memory utilization, to quantify the architecture-aware pass selection and its effect on the earlier metrics. The selected results are summarized in Table 6. On GeForce, where the peak memory utilization remained almost identical for all the experiments, a 10% decrease in the median memory utilization is reported in the SSD_ResNet and a 5% in ResNet50 PyTorch implementation. Power consumption exhibited equivalent behavior. On A100, memory utilization was reduced by 2%-6% across all the runs. The observations confirm that the architecture-aware selection of passes impacts memory utilization and power consumption, an essential aspect of computation on resource-constrained edge devices.

6 CONCLUSION AND FUTURE DIRECTIONS

Deep learning graph compilers are attaining attraction in academia and industry. They have evidently proved successful in extracting the parallelism from the computation graph and applying various optimizations, improving the throughput, latency, memory, and energy consumption. There is a need to understand the distinction in the methodology on how the compiler optimization passes are applied to a DL compiler, like TVM. This research focuses on considering neural architecture while deciding on the pass selection. Further, we exhibited how the proposed approach can prune the search space and significantly reduce compile-time. It can be significant for applications heavily dependent on JIT compilation, like edge computing. Also, with an increasing number of passes and the complexity of the computation graph, it is essential to reduce search space to facilitate static rule-based or ML technique-based pass selection.

In the future, we plan to propose an intelligent methodology to optimize the selection procedure on a resource-constrained edge device, emphasizing the metrics like power consumption and device temperature.

ACKNOWLEDGMENTS

This research work is supported in part by the U.S. Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) under agreement number FA8750-15-2-0119. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) or the U.S. Government. This research used resources of the Argonne Leadership Computing Facility (ALCF), which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We would like to thank anonymous reviewers for their valuable comments and suggestions, which helped us improve the manuscript's quality.

REFERENCES

[1] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[2] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive modeling methodology for compiler phase-ordering. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*. 7–12.

[3] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–28.

[4] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[6] Standard Performance Evaluation Corporation. 1995-2022. *SPECint95 Benchmark Suite*. Retrieved 20220208 from https://www.spec.org/cpu95/CINT95/index.html

[7] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. *arXiv:2109.08267* (2021).

[8] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058* (2018).

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Grigori Fursin and Olivier Temam. 2010. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 4 (2010), 1–29.

[12] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. 2018. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. 35–42.

[13] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. 2005. Optimizing general purpose compiler optimization. In *Proceedings of the 2nd conference on Computing frontiers*. 180–188.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[15] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. 2019. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 308–308.

[16] Yuanjie Huang, Liang Peng, Chengyong Wu, Yuriy Kashnikov, Jörn Rennecke, and Grigori Fursin. 2010. Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*.

[17] Michael R Jantz and Prasad A Kulkarni. 2013. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 1–10.

[18] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. 2021. Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning. In *50th International Conference on Parallel Processing Workshop*. 1–6.

[19] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[20] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. 2009. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 1 (2009), 1–36.

[21] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.

[22] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. SSD: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.

[23] Luiz GA Martins, Ricardo Nobre, Joao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. 2016. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–28.

[24] Luiz GA Martins, Ricardo Nobre, Alexandre CB Delbem, Eduardo Marques, and João MP Cardoso. 2014. A clustering-based approach for exploring sequences of compiler optimizations. In *2014 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2436–2443.

[25] Mena Nagiub and Wael Farag. 2013. Automatic selection of compiler options using genetic techniques for embedded software design. In *2013 IEEE 14th International Symposium on Computational Intelligence and Informatics (CINTI)*. 69–74. https://doi.org/10.1109/CINTI.2013.6705166

[26] Ricardo Nobre, Luiz GA Martins, and Joao MP Cardoso. 2015. Use of previously acquired positioning of optimizations for phase ordering exploration. In *Proceedings of the 18th international workshop on software and compilers for embedded systems*. 58–67.

[27] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. 2016. A graph-based iterative compiler pass selection and phase ordering approach. *ACM SIGPLAN Notices* 51, 5 (2016), 21–30.

[28] Ricardo Nobre, Luís Reis, and Joao MP Cardoso. 2018. Compiler phase ordering as an orthogonal approach for reducing energy consumption. *arXiv preprint arXiv:1807.00638* (2018).

[29] NVIDIA. v2021.2.2.0. *Nsight Compute Command Line Profiler*. Retrieved 20220208 from https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

[30] NVIDIA. v2021.3.2.4. *NVIDIA Nsight Systems*. Retrieved 20220208 from https://docs.nvidia.com/nsight-systems/

[31] NVIDIA. v8.0.1. *TensorRT*. https://developer.nvidia.com/tensorrt

[32] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[33] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[34] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[35] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[37] Gaurav Verma, Yashi Gupta, Abid M Malik, and Barbara Chapman. 2021. Performance evaluation of deep learning compilers for edge inference. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 858–865.