Leveraging Neural Graph Compilers in Machine Learning Research for Edge-Cloud Systems

Alireza Furutanpey*, Carmen Walser, Philipp Raith, Pantelis A. Frangoudis, Schahram Dustdar

***Abstract*—This work presents a comprehensive evaluation of neural network graph compilers across heterogeneous hardware platforms, addressing the critical gap between theoretical optimization techniques and practical deployment scenarios. We demonstrate how vendor-specific optimizations can invalidate relative performance comparisons between architectural archetypes, with performance advantages sometimes completely reversing after compilation. Our systematic analysis reveals that graph compilers exhibit performance patterns highly dependent on both neural architecture and batch sizes. Through fine-grained block-level experimentation, we establish that vendor-specific compilers can leverage repeated patterns in simple architectures, yielding disproportionate throughput gains as model depth increases. We introduce novel metrics to quantify a compiler's ability to mitigate performance friction as batch size increases. Our methodology bridges the gap between academic research and practical deployment by incorporating compiler effects throughout the research process, providing actionable insights for practitioners navigating complex optimization landscapes across heterogeneous hardware environments.**

***Index Terms*—Deep Learning, Neural Networks, Compilers, Graph Optimization, Benchmarking**

I. Introduction

THE pervasiveness of neural networks (NNs) in modern computing systems has generated significant demand for methods to improve the efficiency of available hardware. As computational complexity increases and deployment scenarios diversify, optimizing neural network execution becomes indispensable for practical applications across various computational platforms. Among the most promising optimization approaches are graph compilers, which optimize the computational graphs of neural networks to enhance scheduling, improve data flow, and exploit dedicated hardware modules. Graph compilers can enhance throughput by orders of magnitude with no loss in accuracy. While these compilers can be used independently, they may also be combined with model compression or acceleration methods, such as quantization, that trade off efficiency for accuracy. The potential performance improvements are substantial. Yet, fully leveraging graph compilers presents distinct challenges. Graph compilers and other vendor-specific optimizations can completely alter the relative performance across competing architectures. Figure 1 not only precisely exemplifies this behavior, but also demonstrates that the exact inverse holds for a different device-compiler pair. The models are comparably large, and the batch size is 8. Section V will detail experiment configurations.

All authors are with the Distributed Systems Group, TU Wien, Austria.
*Corresponding author: a.furutanpey@dsg.tuwien.ac.at

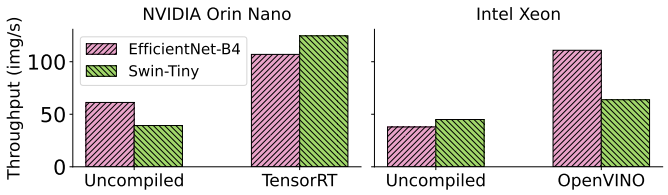


Fig. 1: On the Orin, the convolutional-based EfficientNet has higher throughput than the transformer-based Swin. Applying TensorRT significantly improves throughput for both models. However, the EfficientNet is now slower than the Swin. On the Xeon with OpenVINO, we observe the exact inverse behavior.

Arguably, the increasing complexity of optimizing neural networks creates a disconnect between academic research and real-world applicability, despite directly addressing practical problems. When designing novel machine learning algorithms for Edge(-Cloud) Systems [1], it is crucial to understand how graph compilers can invalidate relative performance differences between architectural archetypes. A common problem when extending research work into real-world systems is determining whether reported performance improvements, regarding resource usage or throughput, from the latest advancements will generalize to the target hardware. This problem stems not from a lack of rigor by researchers but from the inherent heterogeneity of the AI accelerator landscape [2]. This insight was a key motivation in our previous works [3]–[5], where we deliberately opted for simplified encoder architectures with widely supported operations to ensure that reported results would generalize across vendors. While these and similar research contributions are valuable, their practical application requires further consideration, often creating a needlessly high barrier for practitioners by having to navigate complex optimization landscapes. To narrow the gap between research contributions and their applications in real systems, we introduce an automated tool that integrates with existing profilers commonly used in edge or cloud frameworks (e.g., for model selection [6]). The tool streamlines graph compiler benchmarking over heterogeneous compute infrastructure to facilitate iterative empirical analysis. We propose a methodological approach to utilize such tools for designing, implementing, and deploying experiments in research for Edge or Edge-Cloud systems that rely on neural components. We perform a comprehensive empirical analysis across varying architectural families on a heterogeneous physical testbed, demonstrating how vendors prioritize optimizing different layer compositions using both vendor-agnostic (e.g., Apache

TVM) and vendor-specific graph compilers (e.g., TensorRT, OpenVINO). In summary, this work's main contributions are:

- Evaluating graph compilers across heterogeneous hardware platforms, demonstrating the advantages of vendor-specific optimizations.
- Analyzing batch parallelization efficiency across architectures and compilers, revealing optimization opportunities in resource-constrained environments.
- Demonstrating through block-level experimentation that vendor-specific compilers leverage repeated patterns in simpler compositions for disproportionate throughput gains with increased depth.

We are aware that no shortage of work examining the NN graph compiler landscape exists (Sections II and III). However, to the best of our knowledge, this work is the first to discuss their methodological application (Section IV), providing a foundation for informed decision-making based on a comprehensive evaluation (Section V) that includes performance metrics, such as batch scaling efficiency, to uncover non-obvious differences between graph compilers.

II. Related Work

Shuvo et al. [7] provide an excellent review on techniques for utilizing AI accelerators, but it is focused on lower-level tricks for a particular class of hardware. Zhou & Yang benchmark TensorRT [8], but only on convolutional architectures. Li et al. [9] provide a broad overview of existing compilers, but only include rudimentary evaluation on behavior in practice. Like our work, Xing et al. examine graph compilers on different hardware (CPUs, GPUs) [10], but the evaluation only considers individual operations and convolutional-based architectures, without addressing important factors, such as batch size, depth, width, etc. Conversely, this work evaluates graph compilers on various networks from varying architectural families and leverages the broad results to draw generalizable insights. The work by Jajal et al. [11] shares similarities in examining computational graph optimization of varying architectural styles and vendors, but the focus is on interoperability, and specifically the issues that may be encountered when converting models to ONNX. The work in [12] also examines computational graph optimization, but more generally focuses on uncovering bugs in the development cycle of systems that train and deploy deep neural networks. The work in [13] shares similarity in advocating for a design strategy that is mindful of the underlying hardware acceleration. Still, it is an entirely qualitative assessment without any empirical analysis. Lastly, Zhang et al. [14], [15] introduce libraries for benchmarking and provide comprehensive results, but include only mobile platforms and do not examine graph compilers.

III. Background

Graph compilers analyze and optimize computational graphs representing neural networks as nodes (operations) and edges (data dependencies). They provide abstraction to lower-level implementation details by converting models from high-level frameworks (PyTorch, TensorFlow) into hardware-agnostic intermediate representations. Moreover, they may apply transformations that improve execution speed and memory efficiency across AI accelerators, and hardware-specific code generation for low-level kernels tailored to target architectures (e.g., CUDA for NVIDIA GPUs, OpenCL for FPGAs).

A. High-level Network Architecture Organization

Figure 2 illustrates how most modern architectures organize layers. Each layer applies a linear transform, normalization,

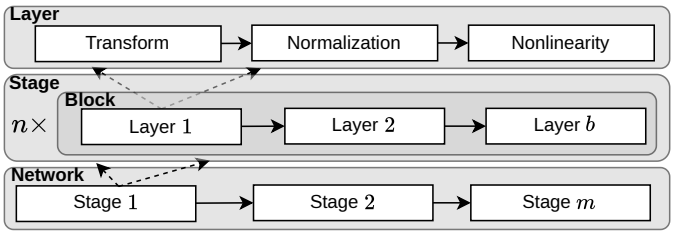


Fig. 2: Network Architecture Layer Organization

and introduces non-linearity with an activation function. Layers are grouped into blocks, which may be more complex, as shown here, such as the ResNet bottleneck [16] that uses two 1×1 convolutional layers to reduce the number of channels, before increasing them again. A stage consists of a sequence of repeated blocks. Finally, an architecture consists of at least one stage, and each stage may have a variable number of blocks. The difference between models of different sizes from the same architecture is typically their width and block ratios. For example, the block ratio in Swin-Tiny is 2:2:**6**:2 and in Swin-Base 2:2:**18**:2 [17]. Graph Compilers can improve throughput by exploiting the repeated patterns present in such organizations and by providing specialized hardware modules for particular compositions (e.g., `Conv-BatchNorm-ReLU`). The following briefly summarizes software and hardware optimizations that compilers commonly use.

B. Software-Level Optimizations

1) Operator Fusion: Operator fusion combines multiple operations, such as convolution and activation, into a single computational kernel. Without operator fusion, each operation would write intermediate results to memory and then read them back for the next operation. By fusing operations, the compiler generates a single kernel that executes all the fused operations sequentially within the same execution context. This eliminates redundant memory accesses and reduces the overhead of launching multiple kernels.

2) Constant Folding: Constant folding identifies subgraphs where all inputs are constants and precomputes them at compile time. This reduces runtime computation by eliminating the need to compute results that do not depend on dynamic inputs repeatedly.

3) Layout Transformation: Different hardware architectures have specific data layout preferences for optimal performance. For example, NVIDIA GPUs with Tensor Cores prefer the BHWC (batch size, height, width, channels) format over BCHW (batch size, channels, height, width). Layout transformations reorganize tensor data into these preferred formats during compilation. These transformations ensure that memory accesses are coalesced and aligned with hardware requirements, improving throughput.

C. Hardware and Kernel-Level Optimizations

1) Kernel Fusion: Kernel fusion is similar to operator fusion, but at the kernel level. Similarly to operator fusion, kernel fusion combines multiple operations into one kernel execution to reduce kernel launch overheads. However, kernel fusion operates at a lower level and can merge operations with finer granularity.

2) Memory Latency Hiding: Memory latency hiding overlaps computation with data transfers using asynchronous execution techniques. Specifically, by overlapping data movement (e.g., between global memory and shared memory) with computation, the memory access latencies appear instantaneous, i.e., "hidden." Similarly to dynamic batching, it involves a static code analysis and code generation for execution paths. The execution paths facilitate asynchronous memory transfers and efficient scheduling of threads, such that some threads perform computations while others wait for data transfers to complete. For example, in matrix multiplication on GPUs, while one block of threads computes partial results using data already loaded into shared memory, another block loads the next set of data from global memory asynchronously.

3) Sparse Computation: Sparse computation exploits sparsity in weights or activations. It leverages tensor sparsity patterns (e.g., weights with many zero values) to skip unnecessary calculations and reduce storage requirements and memory use. In particular, specialized sparse matrix formats like Compressed Sparse Row or Block Sparse Row store only non-zero elements efficiently. Hardware accelerators often include optimized sparse matrix multiplication routines that exploit these formats. For example, consider a sparse neural network where 70% of weights are zero due to unstructured pruning. Instead of performing dense matrix multiplication on all elements, sparse matrix multiplication algorithms process only non-zero elements stored in CSR format. Then, multiplying an input vector with a sparse weight matrix skips zero-weighted connections, reducing computation time and memory bandwidth usage.

IV. Graph Compiler-guided Solution Approach

We implement *NGraphBench*, a library that permits quick, automated, empirical evaluation of graph compilers in a heterogeneous cluster. However, the focus of the work is not the implementation details of the library, and we only mention high-level details for evaluation transparency in Section V. Instead, the focus is on effectively utilizing empirical compiler benchmark results to iteratively conceive and refine ML methods, with a clear application focus on edge-cloud systems.

A. NGraphBench Library

NGraphBench exposes a uniform interface for accessing and integrating graph compiler APIs. Users can provide their models in ONNX or native PyTorch and configure experiments, such as compiler-device pairs, compiler flags, repetitions, and model initialization parameters. Figure 3 illustrates the high-level application flow. The client will deploy the benchmarking application to the requested devices in the cluster. Crucially, the compilation is done *locally* on the target devices, i.e., we

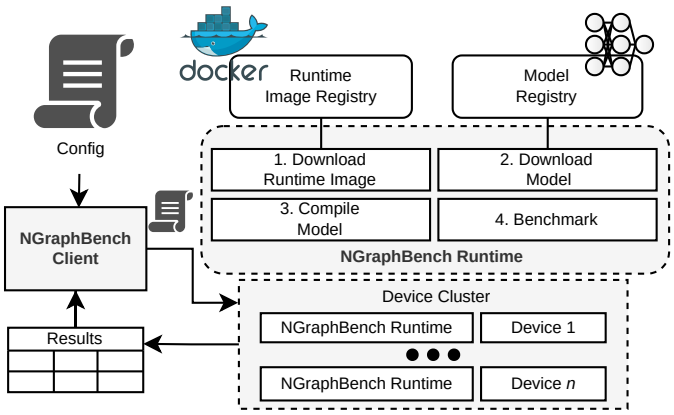


Fig. 3: NGraphBench high-level flow. The client coordinates the experiments for participating clients in a cluster. Each device may evaluate multiple compilers.

are *not* using hardware simulators, which are likely to result in worse optimizations. Each device may benchmark multiple compilers, and will persist results in predefined checkpoints periodically (e.g., to resume on a crash). After benchmarking, the devices will report the results to the client. Once all devices have reported their results, the client will tear down the benchmarking environments and terminate the application.

B. Pragmatic Research Design for Practical Systems

The disconnect between academic research and practical deployment is particularly problematic when optimizing neural networks for heterogeneous hardware. While novel architectures may excel in controlled benchmarks, their performance can vary dramatically when deployed with different graph compilers across diverse hardware. We propose a methodological framework that incorporates compiler effects throughout the research process, as illustrated in Figure 4. This framework divides the research process into three phases, each integrating compiler optimization considerations:

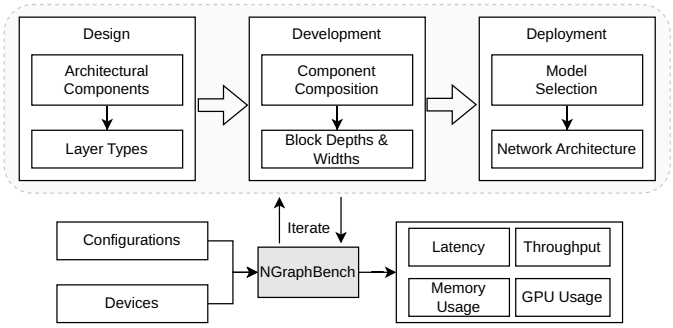


Fig. 4: Iterative refinement guided by empirical analysis

1) Design Phase: The design phase prioritizes architectural choices with widespread hardware and compiler support. Using our work in [4] as a case study, we selected variational compression methods for orbital edge computing applications where processing must occur within finite time windows. Rather than optimizing for theoretical metrics like the number of Multiply-And-Accumulate (MAC) operations or

parameter counts, we introduced the *Transfer Cost Reduction per Second (TCR/s)* metric to balance compression efficiency against computational throughput. This approach enabled the evaluation of different architectural paradigms (convolutional vs. transformer-based) against practical deployment metrics.

2) Development Phase: During development, researchers must examine how block compositions affect model performance and compiler-optimized throughput. Our analysis revealed that increasing model depth often yields disproportionate throughput gains when target hardware incorporates vendor-specific compiler support. Similarly, compiler optimizations can effectively mitigate width adjustments that should reduce throughput. Researchers can identify viable block compositions and configurations that maximize performance within deployment constraints through systematic evaluation with graph compilers.

3) Deployment Phase: The final phase involves comprehensive testing on target hardware with appropriate compiler optimizations. In our case study, without graph compiler optimization, increased model width significantly deteriorated batch parallelization efficiency, contradicting development-phase expectations. This resulted in selecting marginally smaller models for constrained devices despite 30% worse compression performance. The cause was what we refer to as the *batch-width scaling friction*. By slightly increasing the convolutional channels (i.e., the width), the TCR/s has significantly dropped due to reduced processing throughput. Such counterintuitive outcomes highlight the critical importance of evaluating compiler-hardware interactions throughout the research process. In short, our methodology aims to bridge the gap between academic innovation and practical deployment by integrating graph compiler considerations into each research phase. While this approach requires additional empirical testing, it ensures that reported performance improvements generalize across deployment scenarios, ultimately producing more valuable contributions for practitioners working with heterogeneous hardware environments.

V. Evaluation

The evaluation is motivated by the three phases of our compiler-guided framework from Section IV-B. We examine: (1) differential compiler support across architectural styles to inform design decisions in Figure 5; (2) depth scaling and batch-width friction mitigation effects to guide component composition in Section V-C, Section V-E; and (3) analyze graph compiler effects on resource usage to reason about unexpected throughput gains or losses, such as from adverse effects by concurrent tasks in Section V-F.

A. Methodology & Experiment Design

We include TensorRT and OpenVINO to represent vendor-specific compilers and Apache TVM to represent vendor-agnostic compilers with hardware-level optimizations. The ONNX and TorchScript runtimes represent a software-level optimization approach. The evaluation exclusively focuses on applying graph compilers without fundamentally altering prediction behavior, i.e., it does *not* consider quantization and other model compression methods. For experiments with a relative measure that relies on a baseline (e.g., speedup factors, BSR), we use the PyTorch dynamic computational graph and refer to it as the *identity*. We repeat each experiment 100 times and report the average with standard deviations. Compilation is performed on the *native hardware* without hardware simulators. The experiments are performed end-to-end by deploying them on a physical testbed cluster using the NGraphBench library (Section IV). The following details the testbed and configurations to facilitate reproducibility. We emphasize that in this work, the NGraphCompiler library is exclusively for convenience and is not required to reproduce our results.

1) Testbed: We implement a physical testbed with relevant specifications summarized in Table I. For clarity, we will refer to Server 1 and Server 2 as "GPU" and "Xeon" respectively, i.e., the chip we compile for and run the neural network on. The Orin Nano uses `Jetpack 6.2`, which is based on `Ubuntu 22.04`. Hence, the other devices use `Ubuntu 22.04 LTS` with Linux kernel version 5.15. We prioritize consistency over using the latest versions. Table II reports the oldest versions installed on the devices.

TABLE I: Testbed Device Specifications

Device	CPU	GPU
Server 1	8x Ryzen 5700G @ 3.80 Ghz (x86)	RTX 4070
Server 2	8x Xeon Skylake @ 3.0 Ghz (x86)	N/A
Orin Nano	6x Cortex-A78 @ 2.0 Ghz (ARM)	Amp. 512 CC 16 TC

TABLE II: Library Versions

Library	Version	Library	Version
ONNXRuntime	1.19.2	PyTorch	2.4.1
TensorRT	10.4.0	CUDA	12.5
ApacheTVM	0.18.dev0	cuDNN	9.3.0
OpenVINO	2024.3.0	timm	1.0.15

2) Compiler Configurations: Except for Apache TVM, we use intuitive default configurations for graph compilers (e.g., optimize for throughput instead of latency in OpenVINO when the evaluation criterion is throughput). To remain vendor-agnostic, TVM takes a fundamentally different approach to optimization than vendor-specific compilers. TVM can fuse arbitrary patterns and support new operations, if it can find them [18]. Vendor-specific frameworks compile fast, but are limited to pre-defined fusion patterns or operations. TVM's tuning involves running many candidate kernels on the hardware or a simulator to measure performance, yielding highly optimized code, potentially matching or exceeding vendor libraries. The caveat is that TVM traverses an exponentially scaling search space, such that finding an *optimal* computational graph for a single experiment may take weeks or months. Therefore, we cap the number of trials at 1500 with early stopping after 150 using the `xgb` tuner, as we empirically determined on a subset of models that increasing the number of trials beyond 1500 yields diminishing results. Moreover, we only apply TVM to the off-the-shelf models on the *native hardware* and omit it from the block-level evaluation due to time constraints. Table III shows the compile times for the largest models. Notice that even after limiting the number

TABLE III: Contrasting Compile Times in Seconds

Model	Batch Size	Intel Xeon		GeForce RTX	
		OpenVINO	TVM	TensorRT	TVM
ResNet-101	1	2.249	18,022.663	11.863	57,307.949
	32	4.080	28,327.363	16.089	58,466.694
EfficientNet-B5	1	3.109	36,241.423	53.332	69,364.992
	32	5.646	49,765.446	68.360	61,700.123
ConvNeXt-Base	1	4.830	36,431.073	10.004	16,533.937
	32	6.763	68,943.533	19.394	18,404.425
DeiT-Base	1	4.764	11,464.397	6.320	5230.073
	32	6.264	36,199.033	13.827	6157.337
Swin-Base	1	9.338	88,095.557	17.545	27,378.831
	32	12.628	13,4262.784	29.204	14,287.950

of trials, the compilation time may take more than 19 hours. Lastly, notice that the compilation time increases with batch size only for Apache TVM. Unlike OpenVINO and TensorRT, the TVM search heuristic relies on real measurements for each candidate graph, where the runtime scales with the batch size.

3) Network Architecture & Layer Composition: We perform experiments on off-the-shelf architectures and more fine-grained blocks. Evaluating widespread models yields general insights, such as whether vendors favor a particular architectural style. Table IV summarizes the architecture specifications. We use the `timm` [19] library that ensures consistent implementations to access off-the-shelf architectures, so exact parameter and MAC counts may differ slightly from those reported in original publications. We consider five architectural families and three consecutively increasing model sizes per family. We include two convolutional-based (ResNets [16], EfficientNets [20]) and two transformer-based (Swins [17], DeiTs [21]). Additionally, we include ConvNeXts [22] as a hybrid approach that is a convolutional-based model but includes design principles from transformers. Table V summarizes the per-block specifications. Blocks allow us a more targeted evaluation of depth (i.e., investigate optimization as we stack repeated blocks) and width, and reduce noise from certain implementation quirks or other factors that affect compiler efficacy. The multi-head attention (MHA) block uses ReLU nonlinearity. We use channels in convolutional blocks and embedding dimensions for MHA blocks to parameterize block widths when investigating batch-width friction. We found that varying kernel and input sizes similarly affect batch-width friction as increasing the channels. To simplify, we only report results with the kernel size fixed at 3×3 and input size $3 \times 244 \times 244$. In MHA block experiments, we fix the sequence length to ten for the input tensor and consider the embedding dimensions to parameterize the block width.

TABLE IV: Network Architecture Specifications

Architecture	Style	Parameters	MACs
ResNet-18	Convolutional	11,689,512	1,814,083,944
ResNet-50	Convolutional	25,557,032	4,089,238,376
ResNet-101	Convolutional	44,549,160	7,801,511,784
EfficientNet-B3	Convolutional	12,233,232	962,729,320
EfficientNet-B4	Convolutional	19,341,616	1,503,740,472
EfficientNet-B5	Convolutional	30,389,784	2,356,534,504
DeiT-Small	Transformer	22,059,496	79,557,352
DeiT-Medium	Transformer	38,849,512	115,513,320
DeiT-Base	Transformer	86,585,320	201,581,032
Swin-Tiny	Transformer	28,328,674	52,152,040
Swin-Small	Transformer	49,737,298	66,312,424
Swin-Base	Transformer	71,125,762	94,739,176
ConvNeXt-Tiny	Hybrid	28,589,128	322,371,592
ConvNeXt-Small	Hybrid	50,223,688	411,391,240
ConvNeXt-Base	Hybrid	88,591,464	646,530,408

TABLE V: Per-Block Specifications

Convolutional		Multi-Head Attention	
Channels	Params Per Block	Embedding Dimensions	Params Per Block
64	37,056	128	66,048
96	83,232	256	263,168
128	147,840	384	591,360
256	590,592	512	1,050,624

4) Measuring Batch Parallelization: We can measure the Relative Throughput Rate (RTR) as

$$RTR_c(b) = \frac{T_c(b)}{T_c(1)}$$

where $T_c(b)$ is the throughput in samples per second for batch size b when compiler c is used. The RTR quantifies the unnormalized parallelization rate. When scaling is perfect, the throughput linearly scales as a function of the batch size. Note that once RTR drops below 1, the batching reduces absolute throughput. Moreover, perfect scaling does not happen in practice for larger batch sizes, so we will visualize the decay in batch scaling efficiency with the Absolute Scaling Efficiency (ASE) measure:

$$ASE_c(b) = \frac{T_c(b)}{b \cdot T_c(1)}.$$

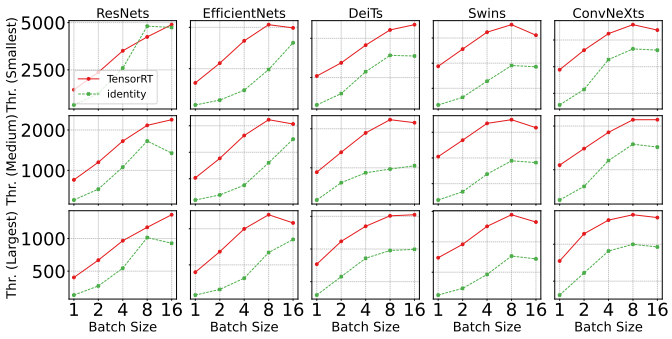
The ASE normalizes the RTR, so any reduction from a perfect parallelization rate directly indicates reduced scaling efficiency. For example, when batch parallelization is perfect, the ASE stays consistently 100%, irrespective of the batch size. Conversely, decreasing ASE implies diminishing returns from increasing the batch size. As discussed in Section IV-B, it is interesting to see whether compilers can mitigate the decay, i.e., maintain scaling efficiency at higher batch sizes. We measure this with the Batch Scaling Resilience (BSR) as follows:

$$BSR_c(b) = \frac{ASE_c(b)}{ASE_{identity}(b)}.$$

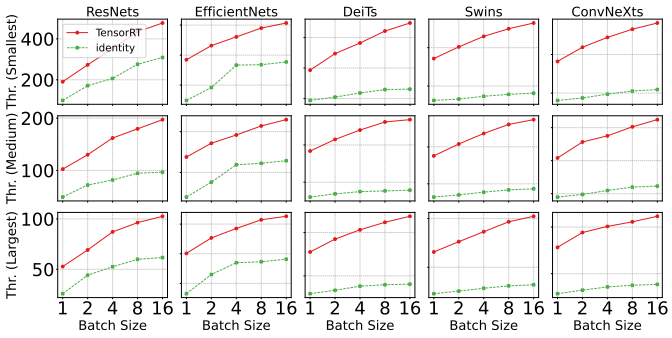
The BSR is a relative measure that can quantify, without eyeballing, the improvement in mitigating friction compared to a baseline compiler. Compilers with BSR values > 1 consistently across varying configurations demonstrate that they can improve the batch scaling efficiency for target hardware. Additionally, measuring BSR as we increase the batch size for different width configurations can determine whether a compiler can alleviate the scaling friction.

B. Differential Compiler Support Across Architectural Styles

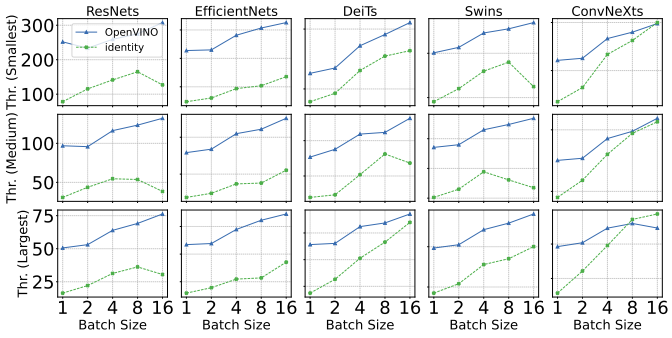
Figure 5 compares the absolute throughput of the vendor-specific compilers with the dynamic uncompiled graphs. Each row corresponds to a model size, and each column to an architectural family (e.g., smallest for ResNets is ResNet-18). The y-axis scaling is non-uniform to highlight the strong relationship between compiler efficacy and architectural style. Notice that even if the absolute throughput across model sizes



(a) Device = GPU



(b) Device = Orin



(c) Device = Xeon

Fig. 5: Contrasting the absolute throughput of a vendor-specific compiler with the baseline uncompiled graph. Compiling results in significant throughput gains, except when resources are scarce, the performance saturates.

is offset, the throughput scaling as we increase the batch size is strikingly similar within a family. The caveat is that the varying device capacities obfuscate the results, making it challenging to assess compilation efficacy. Hence, we report relative values for the remainder of the evaluation but summarize partially aggregated measurements using absolute values in Table VI. Figure 6 plots the throughput multiplier for the five architectural families on different devices-compiler pairs.

The results reveal strikingly distinct performance patterns across batch sizes and neural network architectures, demonstrating performance patterns highly dependent on batch size and neural architecture. Interestingly, despite being advertised as a "convolutional architecture", ConvNeXt behaves similarly to transformers in performance across all compilers. At small batches ($\leq$2), compilers provide 2-6$\times$ speed-ups by eliminating Python dispatch overhead and enabling operation fusion. These advantages diminish as batch size increases, with only vendor-specific solutions (TensorRT for GPU, OpenVINO for CPU) maintaining consistent performance advantages at batch size 16. Architecture significantly influences compiler efficacy: traditional convolutional networks benefit substantially from all compilers at small batches, while transformer-based models show minimal improvement with TVM, moderate gains with ONNX, and substantial acceleration only with vendor-specific tools. While TVM demonstrates substantial performance gains for convolutional architectures, the performance completely tanks for transformer architectures, particularly for ConvNeXt. TVM struggling with ConvNeXt's hybrid design elements is intuitive given its search-based [18] optimization methodology. Conversely, Vendor-specific compilers maintain their advantage through extensive manual optimization targeted specifically at popular cutting-edge architectures. Since the dynamic computational graph is executed sequentially in the Python runtime on the Xeon CPU, we also include TorchScript as an additional baseline. TorchScript can exploit the multiple cores on the CPU with intra-op parallelism. However, compared to OpenVINO, which achieves significant speed-ups (up to 5-6 times more throughput), TorchScript only marginally improves throughput across all configurations.

TABLE VI: Aggregated Results by Architectural Family

Family	Compiler	Batch Size(2-4)		Batch Size(8-16)	
		Throughput [t/s] ↑	CPU [%] ↓	Throughput [t/s] ↑	CPU [%] ↓
ResNets	Identity (Orin)	104.69 ± 0.57	11.13 ± 0.78	149.63 ± 0.66	5.52 ± 2.00
	Identity (GPU)	1047.32 ± 14.09	6.24 ± 0.09	2442.30 ± 43.44	6.22 ± 0.10
	Identity (CPU)	68.08 ± 1.75	99.83 ± 0.94	75.12 ± 2.78	95.41 ± 2.60
	TensorRT (Orin)	179.01 ± 6.98	3.45 ± 1.14	247.99 ± 1.28	1.58 ± 0.77
	TensorRT (GPU)	1740.11 ± 14.18	6.26 ± 0.14	2673.77 ± 19.56	6.24 ± 0.03
	OpenVINO (CPU)	136.94 ± 5.80	97.98 ± 0.84	163.90 ± 6.21	96.99 ± 1.40
	TVM (GPU)	1678.78 ± 29.22	6.32 ± 0.17	1823.83 ± 374.62	6.26 ± 0.05
	TVM (CPU)	46.55 ± 0.69	49.99 ± 0.72	60.27 ± 0.99	49.88 ± 0.83
EfficientNets	Identity (Orin)	50.33 ± 0.25	14.29 ± 0.83	64.47 ± 0.17	6.27 ± 3.75
	Identity (GPU)	294.85 ± 4.57	6.25 ± 0.10	1029.42 ± 4.12	6.24 ± 0.08
	Identity (CPU)	30.80 ± 0.92	99.05 ± 1.26	47.34 ± 0.96	97.43 ± 2.90
	TensorRT (Orin)	93.74 ± 0.48	5.31 ± 0.83	113.39 ± 0.36	2.03 ± 0.90
	TensorRT (GPU)	1114.56 ± 8.67	6.23 ± 0.14	1578.73 ± 5.53	6.24 ± 0.17
	OpenVINO (CPU)	96.04 ± 2.35	98.28 ± 0.83	120.00 ± 3.28	97.36 ± 1.32
	TVM (GPU)	1088.48 ± 32.06	6.21 ± 0.12	1261.05 ± 236.20	6.26 ± 0.11
	TVM (CPU)	17.69 ± 0.42	49.96 ± 0.80	20.66 ± 0.22	49.94 ± 0.82
DeiTs	Identity (Orin)	36.45 ± 0.26	6.14 ± 1.39	41.50 ± 0.16	1.89 ± 1.96
	Identity (GPU)	674.76 ± 5.76	6.26 ± 0.14	1015.49 ± 3.17	6.25 ± 0.13
	Identity (CPU)	37.08 ± 1.18	99.59 ± 1.58	50.81 ± 1.55	99.11 ± 1.33
	TensorRT (Orin)	95.67 ± 1.06	1.40 ± 0.84	116.26 ± 0.70	0.60 ± 0.78
	TensorRT (GPU)	1164.57 ± 9.87	6.28 ± 0.15	1531.85 ± 6.62	6.25 ± 0.11
	OpenVINO (CPU)	50.29 ± 1.52	99.01 ± 1.07	61.01 ± 2.27	97.26 ± 2.43
	TVM (GPU)	71.43 ± 4.02	6.25 ± 0.07	87.22 ± 5.54	6.25 ± 0.09
	TVM (CPU)	7.00 ± 0.06	49.92 ± 0.89	8.56 ± 0.07	49.93 ± 0.85
Swins	Identity (Orin)	22.28 ± 0.12	7.20 ± 2.80	25.66 ± 0.05	3.08 ± 4.38
	Identity (GPU)	304.23 ± 2.38	6.25 ± 0.08	591.81 ± 1.24	6.25 ± 0.09
	Identity (CPU)	21.37 ± 0.56	96.62 ± 2.52	23.81 ± 0.52	95.29 ± 2.75
	TensorRT (Orin)	68.00 ± 0.55	1.25 ± 0.74	81.62 ± 0.84	0.56 ± 0.88
	TensorRT (GPU)	918.61 ± 7.71	6.22 ± 0.23	1062.34 ± 3.32	6.25 ± 0.14
	OpenVINO (CPU)	36.75 ± 0.81	99.26 ± 0.98	42.79 ± 1.19	98.01 ± 1.90
	TVM (GPU)	149.52 ± 15.19	6.25 ± 0.09	80.34 ± 3.27	6.27 ± 0.17
	TVM (CPU)	8.01 ± 0.17	49.94 ± 0.81	9.67 ± 0.08	49.90 ± 0.85
ConvNeXts	Identity (Orin)	29.72 ± 0.19	6.51 ± 1.92	34.34 ± 0.08	2.32 ± 2.98
	Identity (GPU)	563.15 ± 5.28	6.24 ± 0.17	862.64 ± 3.24	6.24 ± 0.10
	Identity (CPU)	31.86 ± 1.99	98.31 ± 3.28	45.00 ± 1.33	99.37 ± 1.81
	TensorRT (Orin)	76.98 ± 0.57	2.21 ± 0.97	89.80 ± 0.43	0.74 ± 0.82
	TensorRT (GPU)	972.13 ± 6.93	6.25 ± 0.04	1166.44 ± 3.43	6.26 ± 0.12
	OpenVINO (CPU)	38.91 ± 0.91	99.43 ± 0.94	45.50 ± 1.29	97.13 ± 1.91
	TVM (GPU)	266.40 ± 24.91	6.26 ± 0.10	143.74 ± 17.35	6.25 ± 0.08
	TVM (CPU)	10.67 ± 0.21	49.95 ± 0.80	13.50 ± 0.12	49.91 ± 0.87

The results show strikingly distinct performance patterns across compilers, hardware platforms, and neural architectures. Convolutional-based networks benefit from all compilers, while transformer-based models see limited gains from vendor-agnostic solutions. Vendor-specific compilers maintain advantages through targeted optimization for popular architectures, while automated tuning approaches struggle with hybrid designs.

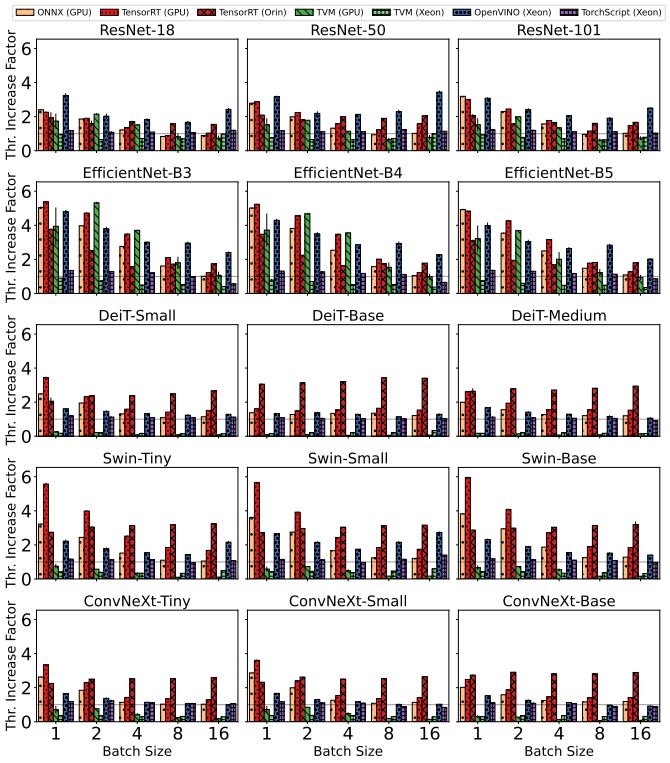
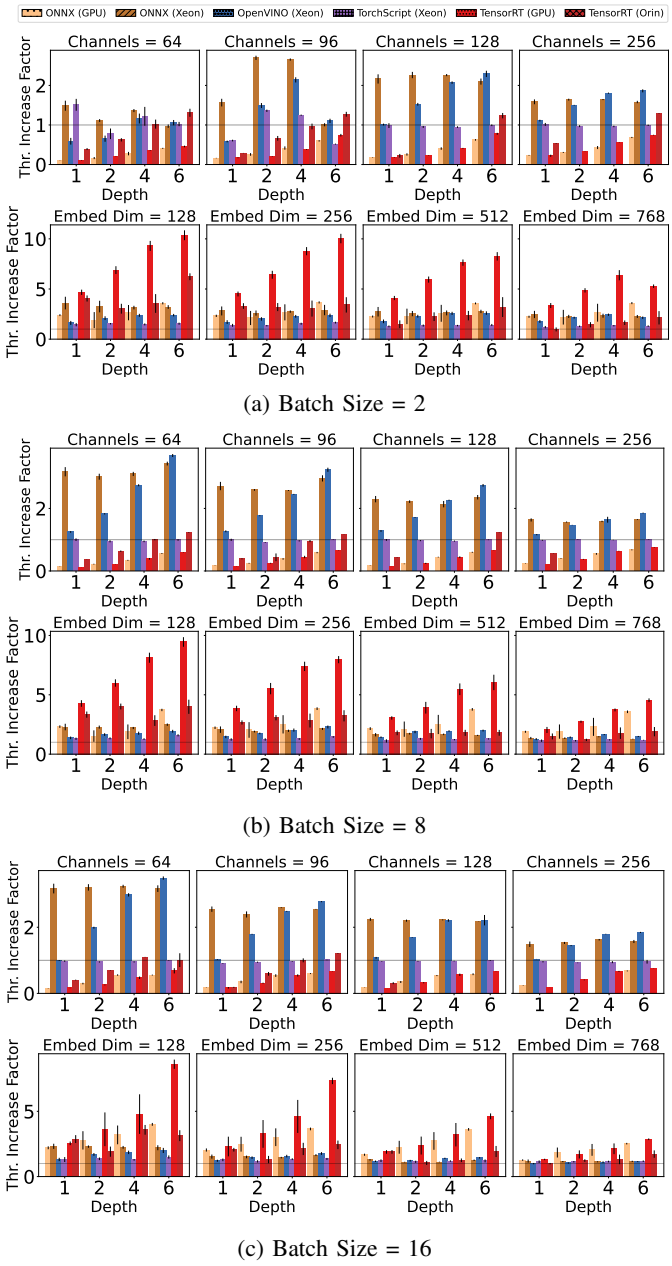


Fig. 6: Contrasting multiplicative throughput gain relative to the baseline. Conv-based architectures see broader support, whereas for transformer-based architectures, it strongly varies.

C. Exploiting Repeated Patterns from Depth Scaling

Figure 7 plots the throughput multiplier for the convolutional and MHA blocks separately. Note that the Y-axis scaling is non-uniform to accentuate the relationship between compiler-device pairs at a set batch size. Unsurprisingly, we observe comparable behavior of convolutional blocks and MHA blocks as with convolutional-based and transformer-based architectures. However, the depth noticeably impacts the throughput relationship between the compilers. This trend remains even when performance starts to saturate due to high computational load from large batch sizes and block widths. For example, for the MHA block at batch size 8 with embed dimension 128, using the TensorRT compiler on the GPU, the throughput is roughly twice that of the baseline dynamic computational graph on a single block, but jumps to eightfold when stacking six repeating blocks. Table VII and Table VIII quantify how varying compilers can leverage the repeating patterns. We compute the *slope* with the least-squares fit to measure how much speed-up changes as we increase the depth. A positive slope implies that each additional layer makes the compiler's advantage even larger. In contrast, a negative slope means that extra blocks diminish initial gains. The *retention* is simply the ratio between the speed-up factor with the deepest stack and the speed-up factor with a single block. A value close to 1 implies that a compiler is agnostic towards the depth parameter, i.e., it cannot leverage the repeated block patterns. Values above 1 imply that the compiler can leverage repeated blocks. For convolutional



(a) Batch Size = 2

(b) Batch Size = 8

(c) Batch Size = 16

Fig. 7: Contrasting how stacking homogeneous blocks improves throughput over the uncompiled graph. The relative relationship between depth and the factor remains consistent.

blocks, the speed-ups of the vendor-specific compilers with hardware optimization are *amplified* by increasing the depth. The convolutional blocks are a repeated sequence of the simple `Conv` →`BatchNorm` →`ReLU` pattern. The simple design aids the hardware-based compilers. The vendor-specific compilers can leverage the repeating patterns and tile and fuse them into smaller or larger kernels. For example, this can occur by collapsing all 3N pointwise ops into one fused pass, or merging multiple 2D convolutions into one multi-stage convolution that reuses intermediate results. As we increase the depth, the *amortized* overhead per block of kernel launch, memory barriers, and descriptor setup decreases. Conversely, the software-based optimization of ONNX shows weaker

TABLE VII: Depth Scaling of Convolutional Blocks

Device	Compiler	Width	Batch Size 8		Batch Size 16	
			Slope	Retention	Slope	Retention
GPU	TensorRT	64	0.08924	3.821	0.1032	1.187
GPU	TensorRT	96	0.1024	0.2116	0.1038	0.6593
GPU	ONNX	64	0.06953	1.153	0.09141	0.426
GPU	ONNX	96	0.07876	1.304	0.08367	0.7538
Orin	TensorRT	64	0.1695	0.7846	0.1349	0.7759
Orin	TensorRT	96	0.1285	1.126	0.212	6.553
Xeon	ONNX	64	0.04862	1.009	0.0006561	0.9756
Xeon	ONNX	96	0.0453	1.025	0.02658	0.9954
Xeon	OpenVINO	64	0.4417	0.3379	0.4775	0.7428
Xeon	OpenVINO	96	0.3553	0.8669	0.3378	0.4106

improvements. The retention values of MHA blocks are, on

TABLE VIII: Depth Scaling of MHA Blocks

Device	Compiler	Width	Batch Size 8		Batch Size 16	
			Slope	Retention	Slope	Retention
GPU	TensorRT	128	0.8207	1.404	1.028	0.556
GPU	TensorRT	256	0.6369	0.6935	0.8831	1.471
GPU	ONNX	128	0.3594	2.187	0.3106	0.83
GPU	ONNX	256	0.3406	1.724	0.2929	0.9108
Orin	TensorRT	128	0.05291	1.178	0.173	0.9055
Orin	TensorRT	256	0.08368	0.95	0.1488	1.629
Xeon	ONNX	128	0.04244	1.105	-0.01775	1.018
Xeon	ONNX	256	0.01867	1.031	0.0215	0.9735
Xeon	OpenVINO	128	0.1113	1.115	0.1355	0.9635
Xeon	OpenVINO	256	0.1589	1.577	0.1015	1.029

average, less than for convolutional blocks, i.e., the compilers cannot leverage the repeated patterns as effectively. The dependency graph of the `Linear(QKV)` →`Reshape` →`MatMul` →`Softmax` →`MatMul` →`Add` →`LayerNorm` →`ReLU` is significantly more complex, such that we may get good one-block kernels, but stacking them does not result in intra-block fusion.

The results reveal that the increased efficiency of using repeated patterns of simple layers may outweigh the benefits of more sophisticated but complex layer types in resource-constrained environments.

D. Batch Parallelization Scaling Efficiency

From both Figure 6 and Figure 7, it is apparent that increasing the batch size significantly influences the throughput rate, and different compilers exhibit varying behavior. Figure 8 illustrates explicitly how increasing the batch size decreases the scaling efficiency despite increasing the raw throughput. Apache TVM shows considerable but inconsistent scaling efficiency for convolutional-based architectures on the GPU. This is expected due to TVM's search-heuristic-based optimization, i.e., we must start a new search for each batch size. Conversely, the scaling efficiency decay of TensorRT is more predictable, as it is consistent with negligible variance.

To account for varying compute capacities, and to provide information on relative improvement over the dynamic computational graph baseline, Figure 9 plots the batch scaling resilience (Section V-A4). Note that a BSR below 1 implies steeper efficiency losses relative to the baseline. We argue that a BSR below 1.0 indicates that there are potentially further opportunities for optimization that the compiler has missed. The intuition is that if the compiler has found a global maximum, the drop in scaling efficiency should be *at*

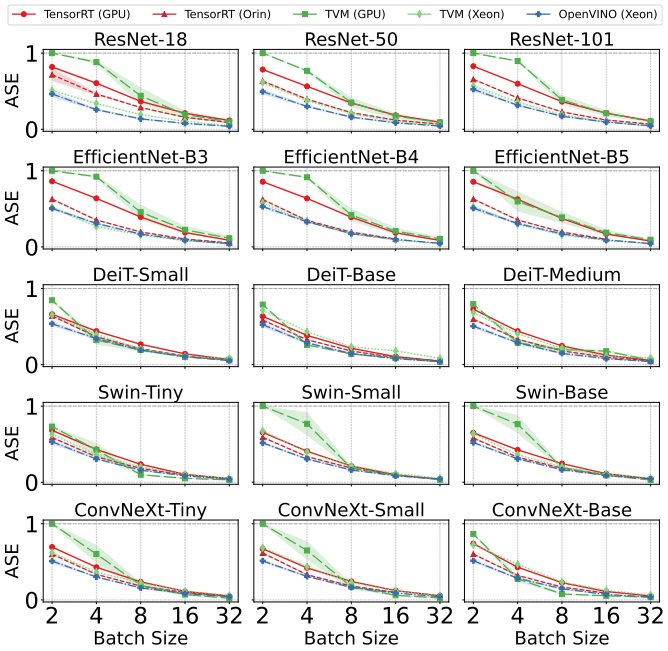


Fig. 8: Contrasting ASE (higher is better) of architectural styles. As performance saturates at higher batch sizes, the throughput gain advantage from parallelization diminishes.

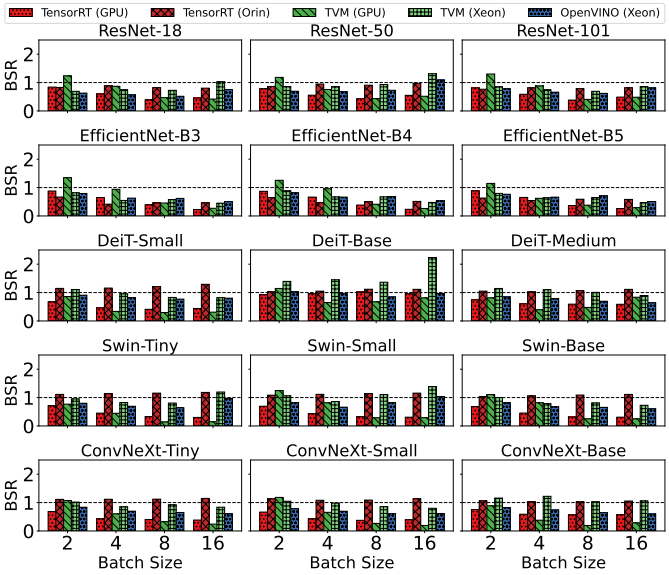


Fig. 9: Contrasting BSR (higher is better) of architectural styles. TensorRT has BSR $\approx$ 1 across most architectures, which implies consistent throughput gains over the baseline.

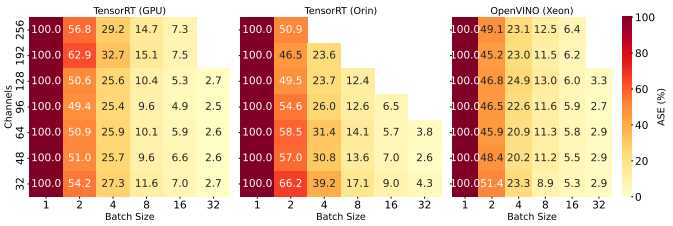
worst consistent between the unoptimized dynamic and the optimized compiled computational graph. In particular, the erratic results of TVM demonstrate that specific optimizations exist that the corresponding vendor does not adequately consider, but they are challenging to find. For example, applying TVM to the mid-sized DeiT TVM shows significantly higher resilience than OpenVINO on the Xeon CPU. Conversely, the resource-constrained Orin shows a BSR of roughly 1.0 across all transformer-based architectures and sizes while

showing substantial throughput gains for the same architecture. However, especially for smaller architectures, the BSR of TensorRT on the powerful GPU is consistently below 1.0.

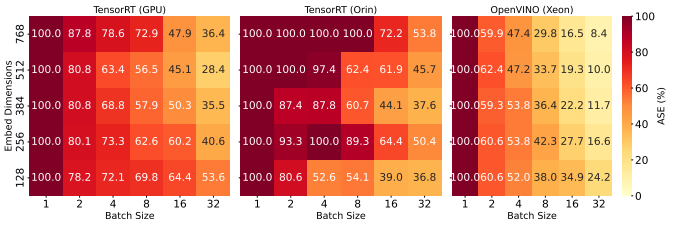
The Batch Scaling Resilience (BSR) metric uncovers compiler-specific optimization patterns, demonstrating that TensorRT achieves consistent scaling profiles (BSR ≈ 1) for most architectures while TVM shows erratic but occasionally superior resilience for specific model-hardware combinations. The results show that compilers can more easily optimize for resource efficiency when resources are scarce. When resources are abundant, BSR values below 1.0 suggest that further optimizations are possible.

E. Batch-Width Scaling Friction Mitigation

We investigate whether compilers can improve with block-level experiment for the same reasons as Section V-C. The



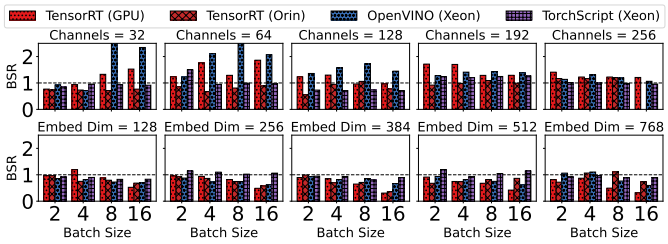
(a) Convolutional Blocks (Depth=6)



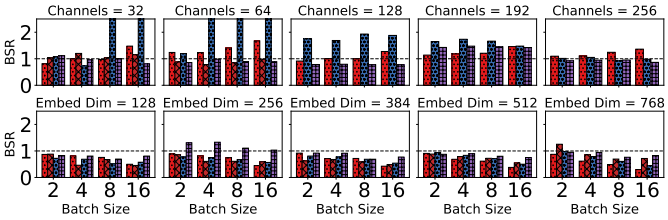
(b) MHA Blocks (Depth=6)

Fig. 10: Heatmaps plotting the effect of width on ASE. The scaling efficiency decreases faster for wider networks, but the rate varies.

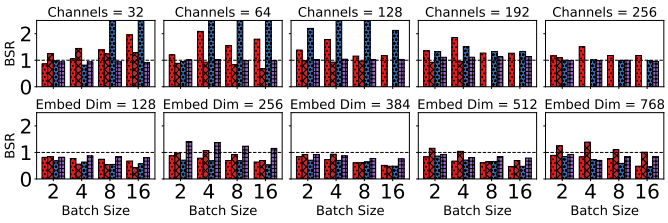
batch-width friction is directly apparent from Figure 10. As we increase the width for a block, the efficiency scaling drops considerably faster from one batch size to the next larger batch size. However, to account for hardware differences and to compare with the baseline dynamic computational graph Figure 11c plots the BSR for three depth configurations. Increasing the depth tends to moderately improve BSR for TensoRT, arguably for the same reasons as outlined in Section V-C. TorchScript slightly mitigates the scaling friction for some configurations through intra-ops parallelization, which is expected. A BSR value higher than one implies that scaling efficiency decreases more gracefully relative to the uncompiled dynamic graph. This is best seen with OpenVINO. For the convolutional blocks, it can considerably mitigate the efficiency decrease by exploiting the multiple cores.



(a) Depth = 1



(b) Depth = 3



(c) Depth = 6

Fig. 11: Contrasting BSR between convolutional and MHA blocks. Successful optimization on the CPU shows improved parallelization rates even at higher batch sizes.

Compiler efficacy is significant for batch-width scaling friction, and the BSR metric accentuates whether compilers can maintain scaling efficiency as width increases. In particular, OpenVINO demonstrates superior mitigation for convolutional blocks through multi-core utilization.

F. Resource Usage Reduction

From Table VI it is apparent that when compilers successfully optimize the graph to have considerable throughput gains, the CPU usage increases on the Xeon where there is no dedicated GPU. On the GPU server and the Jetson board, TensorRT can decrease the CPU usage - marginally on the powerful server and significantly on the constrained Jetson board. TensorRT on GPU leverages static-graph capture and aggressive kernel scheduling to slash CPU-side launch overhead, which on the constrained Jetson Orin's SoC shows up as CPU-usage drops. On a higher-end GPU Server, these savings are negligible. Figure 12 directly compares the CPU usage of TensorRT on the compiled network and the baseline on all architectural families on the Orin device. Notice that for the transformer-based architectures, particularly for the Swin family, there is up to 80% reduction in CPU usage. This is valuable in constrained environments that perform auxiliary tasks on the CPU. For example, in [4], interference from pre- and post-processing on the CPU was negligible on smaller models but adversely affected the throughput of larger models.

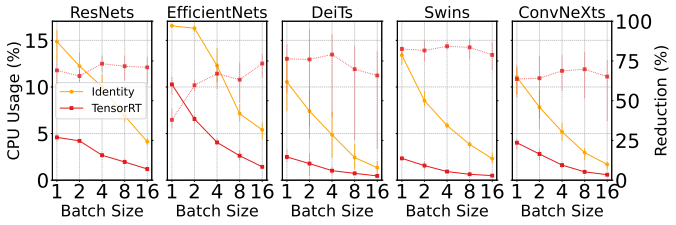


Fig. 12: The left Y-axis (solid line) shows the absolute CPU usage. The left Y-axis (dashed line) shows the relative CPU reduction. When the batch size increases, throughput performance saturates on the GPU, such that the CPU usage reduces.

Applying compilers on devices with a dedicated accelerator can significantly reduce CPU utilization (up to 80%) through static-graph capture and kernel scheduling optimizations. This reduction is particularly valuable in edge computing scenarios where horizontal scaling is limited and CPUs may handle concurrent auxiliary tasks. These findings indicate compiler selection should consider both throughput and resource utilization metrics when deploying neural networks in resource-constrained edge-cloud systems.

VI. Conclusion

The work introduced a framework for incorporating compiler effects throughout the research process for Edge-Cloud systems relying on NNs. Empirical analysis demonstrated that optimizations can completely invalidate performance expectations by systematically analyzing compiler behavior across heterogeneous platforms. The introduced Batch Scaling Resilience metric quantifies a compiler's ability to mitigate performance friction as batch size increases. Block-level experimentation confirmed that simple compositions with widely supported operations provide significant advantages in resource-constrained environments, as compilers effectively leverage repeated patterns for disproportionate throughput gains.

Acknowledgment

We thank Alexander Knoll for providing us with the hardware infrastructure.

References

[1] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[2] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Ai and ml accelerator survey and trends," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–10, IEEE, 2022.

[3] A. Furutanpey, P. Raith, and S. Dustdar, "Frankensplit: Efficient neural feature compression with shallow variational bottleneck injection for mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 10770–10786, 2024.

[4] A. Furutanpey, Q. Zhang, P. Raith, T. Pfandzelter, S. Wang, and S. Dustdar, "Fool: Addressing the downlink bottleneck in satellite computing with neural feature compression," *IEEE Transactions on Mobile Computing*, pp. 1–18, 2025.

[5] A. Furutanpey, P. A. Frangoudis, P. Szabo, and S. Dustdar, "Adversarial robustness of bottleneck injected deep neural networks for task-oriented communication," in *Proc. IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, 2025.

[6] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "Inferline: latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, (New York, NY, USA), p. 477–491, Association for Computing Machinery, 2020.

[7] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023.

[8] Y. Zhou and K. Yang, "Exploring tensorrt to improve real-time inference for deep learning," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pp. 2011–2018, IEEE, 2022.

[9] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2021.

[10] Y. Xing, J. Weng, Y. Wang, L. Sui, Y. Shan, and Y. Wang, "An in-depth comparison of compilers for deep neural networks on hardware," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pp. 1–8, 2019.

[11] P. Jajal, W. Jiang, A. Tewari, E. Kocinare, J. Woo, A. Sarraf, Y.-H. Lu, G. K. Thiruvathukal, and J. C. Davis, "Interoperability in deep learning: A user survey and failure analysis of onnx model converters," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, (New York, NY, USA), p. 1466–1478, Association for Computing Machinery, 2024.

[12] X. Zhang, W. Jiang, C. Shen, Q. Li, Q. Wang, C. Lin, and X. Guan, "Deep learning library testing: Definition, methods and challenges," *ACM Comput. Surv.*, vol. 57, Mar. 2025.

[13] H. Zhang, M. Xing, Y. Wu, and C. Zhao, "Compiler technologies in deep learning co-design: A survey," *Intelligent Computing*, vol. 2, p. 0040, 2023.

[14] Q. Zhang, X. Li, X. Che, X. Ma, A. Zhou, M. Xu, S. Wang, Y. Ma, and X. Liu, "A comprehensive benchmark of deep learning libraries on mobile devices," in *Proceedings of the ACM Web Conference 2022*, WWW '22, (New York, NY, USA), p. 3298–3307, Association for Computing Machinery, 2022.

[15] Q. Zhang, X. Che, Y. Chen, X. Ma, M. Xu, S. Dustdar, X. Liu, and S. Wang, "A comprehensive deep learning library benchmark and optimal library selection," *IEEE Transactions on Mobile Computing*, vol. 23, no. 5, pp. 5069–5082, 2024.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[17] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 10012–10022, 2021.

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: an automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (USA), p. 579–594, USENIX Association, 2018.

[19] R. Wightman, "Pytorch image models." https://github.com/rwightman/pytorch-image-models, 2019.

[20] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[21] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jegou, "Training data-efficient image transformers & distillation through attention," in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 10347–10357, PMLR, 18–24 Jul 2021.

[22] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11976–11986, June 2022.