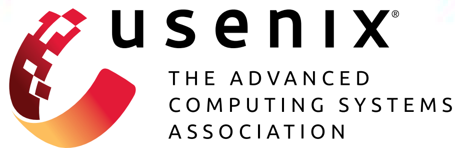


Faith: An Efficient Framework for Transformer Verification on GPUs

Boyuan Feng, Tianqi Tang, Yuke Wang, Zhaodong Chen, Zheng Wang, Shu Yang, Yuan Xie, and Yufei Ding, *University of California, Santa Barbara*

https://www.usenix.org/conference/atc22/presentation/feng

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Faith: An Efficient Framework for Transformer Verification on GPUs

Boyuan Feng, Tianqi Tang, Yuke Wang, Zhaodong Chen, Zheng Wang, Shu Yang, Yuan Xie, and Yufei Ding
University of California, Santa Barbara
{boyuan,tianqi_tang,yuke_wang, chenzd15thu, zheng_wang, shuyang1995, yuanxie, yufeiding}@ucsb.edu

Abstract

Transformer verification draws increasing attention in machine learning research and industry. It formally verifies the robustness of transformers against adversarial attacks such as exchanging words in a sentence with synonyms. However, the performance of transformer verification is still not satisfactory due to bound-centric computation which is significantly different from standard neural networks. In this paper, we propose **Faith**[1], an efficient framework for transformer verification on GPUs. We first propose a semantic-aware computation graph transformation to identify semantic information such as bound computation in transformer verification. We exploit such semantic information to enable efficient kernel fusion at the computation graph level. Second, we propose a verification-specialized kernel crafter to efficiently map transformer verification to modern GPUs. This crafter exploits a set of GPU hardware supports to accelerate verification-specialized operations which are usually memory-intensive. Third, we propose an expert-guided autotuning to incorporate expert knowledge on GPU backends to facilitate large search space exploration. Extensive evaluations show that Faith achieves 2.1× to 3.4× (2.6× on average) speedup over state-of-the-art frameworks.

1 Introduction

Transformers [8, 21, 25, 32, 33, 38, 45] is an important category of neural networks (NNs) in machine learning research and industry. Transformers are first designed for natural language processing (NLP) and have achieved state-of-the-art accuracy across many NLP tasks such as neural machine translation [1, 26, 31] and sentiment analysis [7, 37, 48]. Due to its success, transformers have been widely used in many industrial products such as Facebook for hate speech detection [10] and Alexa for question answering [14]. Recently, transformers also show extraordinary accuracy for many computer vision tasks [9, 19, 44, 47, 55] and become the new trending model.

[1]The project is open-sourced at https://github.com/BoyuanFeng/Faith

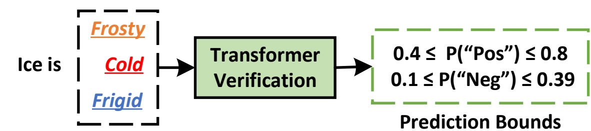


Figure 1: Illustration of transformer verification. Here, all perturbed inputs share the same prediction "positive" since the lower bound probability for "positive" (0.4) is higher than the upper bound probability for "negative" (0.39).

However, similar to prior NNs, transformers are also vulnerable to adversarial attacks that add imperceptible perturbations to input data for maliciously changing transformer predictions [2, 3, 16, 17, 22]. One specific example of adversarial attack is to exchange words (*e.g.*, cold) in a sentence with carefully selected synonyms (*e.g.*, frigid). This vulnerability may result in security concerns for real-world applications. For example, an intentionally crafted hate speech may spread widely on social network.

Transformer verification has been proposed to formally verify the robustness of a transformer against adversarial attacks [4, 18, 35, 42]. Given an input data x and a transformer $F(x)$, transformer verification identifies a maximal bound ε, such that all inputs x' that are "close" to the input data (*i.e.*, $|x'-x| \leq \varepsilon$) cannot "mislead" the transformer (*i.e.*, $F(x) = F(x')$). A larger ε indicates better robustness. Early verification approaches [18] enumerate all possible inputs x' that satisfy $|x'-x| \leq \varepsilon$ and conduct inference on each input to check predictions. These approaches show prohibitive latency due to the large number of inputs x'. Recent transformer verification [35, 42] avoids such enumeration by providing a single pair of lower and upper bounds for transformer predictions over all these inputs, as illustrated in Fig. 1. We can verify the robustness of a transformer if the lower bound of the correct prediction is higher than the upper bound of other predictions. The key computing pattern is a *bound-centric computation*, which computes a pair of inequality bounds for individual neurons. It first represents the input perturbations with inequality bounds over input neurons (*e.g.*, $x-\varepsilon \leq x' \leq x+\varepsilon$) and then propagates these bounds across layers to generate

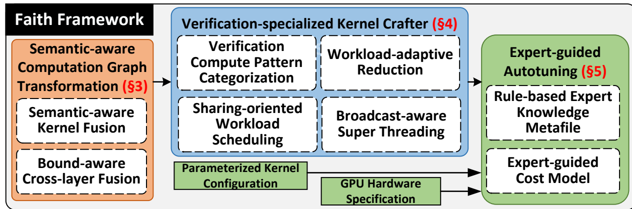


Figure 2: Overview of Faith Framework

the bounds for transformer predictions.

While transformer verification can formally verify the robustness of transformers, it also introduces high latency and limits its applications. In particular, transformer verification usually leads to second-level latency [35] in contrast to millisecond-level latency of standard transformers. We identify three challenges behind efficient transformer verification.

Lack of performance optimization over transformer verification computing patterns. Existing transformer verifications usually utilize the existing deep learning (DL) frameworks, such as PyTorch [30], which are designed for standard NNs. However, transformer verification shows significantly different computing patterns from standard NNs due to the nature of bound-centric computation. For example, when computing the upper bound of an output neuron, transformer verification needs to use the upper bound of the input neuron if the weight is positive; and the lower bound of the input neuron if negative. Straightforwardly deploying transformer verification to the existing DL frameworks usually leads to poor performance.

Lack of framework support for verifying diverse NN layers. Transformer verification shows large diversity in the bound computation for different types of NN layers such as projection layer with only perturbed features and self-attention layer with both perturbed weights and features. Even for the same type of NN layers, diverse upper bounds and lower bounds may be designed which requires different implementations. For example, Crown [52] utilizes two ReLU bound designs for generating more precise bounds for verification, where these bounds are selected dynamically according to the range of input neurons. This diversity makes it challenging to hand optimize GPU kernels in transformer verification.

Lack of verification-specialized adaptability towards modern GPUs. Transformer verification involves abundant memory-intensive operations such as reduction and broadcast. These memory-intensive operations can usually be significantly accelerated with rich architecture supports (*e.g.*, warp-level synchronized reduction) in modern GPUs. However, existing DL frameworks usually only focus on computation-intensive operations (*e.g.*, convolution) and ignore abundant optimization opportunities for memory-intensive operations. This leads to significant overhead in transformer verification with a large number of memory-intensive operations.

In this paper, we build **Faith**, the first framework for efficient transformer verification on GPUs. We show an overview of the Faith framework in Fig. 2. First, we propose ***semantic-aware computation graph transformation*** to fully exploit fusion opportunities in transformer verification at the computation graph level. Our key insight is that transformer verification shows significantly different computing patterns (*e.g.*, two kernels for computing lower and upper bounds involve similar input data) from standard NNs. These computing patterns usually exhibit abundant data reuse opportunities. By exploiting such semantic information, Faith can fully harvest performance potential in transformer verification and achieve significant speedup over existing DL frameworks.

Second, we propose a ***verification-specialized kernel crafter*** to optimize transformer verification towards modern GPUs. Transformer verification contains abundant memory-intensive operations, such as elementwise computation, reduction, and broadcast. These operations may have complex dependencies and lead to performance bottlenecks. To this end, Faith automatically exploits a set of GPU architecture supports to improve the parallelism of such operations. Moreover, Faith introduces a set of optimizations to effectively mitigate memory access and improve performance by exploiting GPU memory hierarchies.

Third, we propose ***expert-guided autotuning*** to efficiently search optimized implementations in the large search space. Existing DL frameworks [6, 54] usually conduct autotuning in a hardware-agnostic approach where an ML-based cost model is deployed to implicitly learn hardware impact over performance from scratch. Instead, we propose a rule-based expert knowledge metafile to explicitly provide a small set of hardware characterizations and an expert-guided cost model to incorporate the expert knowledge. Faith exploits these two components to achieve efficient schedule exploration in the large design space of transformer verification.

In summary, this paper makes the following contributions:

- We build Faith, the first efficient framework to optimize the performance of transformer verification on GPUs.
- We propose a set of verification tailored system optimizations. In particular, we design a *semantic-aware computation graph transformation* to identify and exploit novel fusion opportunities for transformer verification, a *verifier-specialized kernel crafter* to effectively map transformer verification kernels to GPU backends, and an *expert-guided autotuning* to incorporate a set of expert knowledge on modern GPU architecture to guide large design space exploration.
- Extensive experiments show that Faith achieves up to 3.4× speedup (2.6× on average) over state-of-the-art frameworks.

2 Related Work and Motivation

In this section, we first introduce the background of transformer verification (§2.1). Then, we discuss related work on DL frameworks (§2.2). Finally, we present opportunities

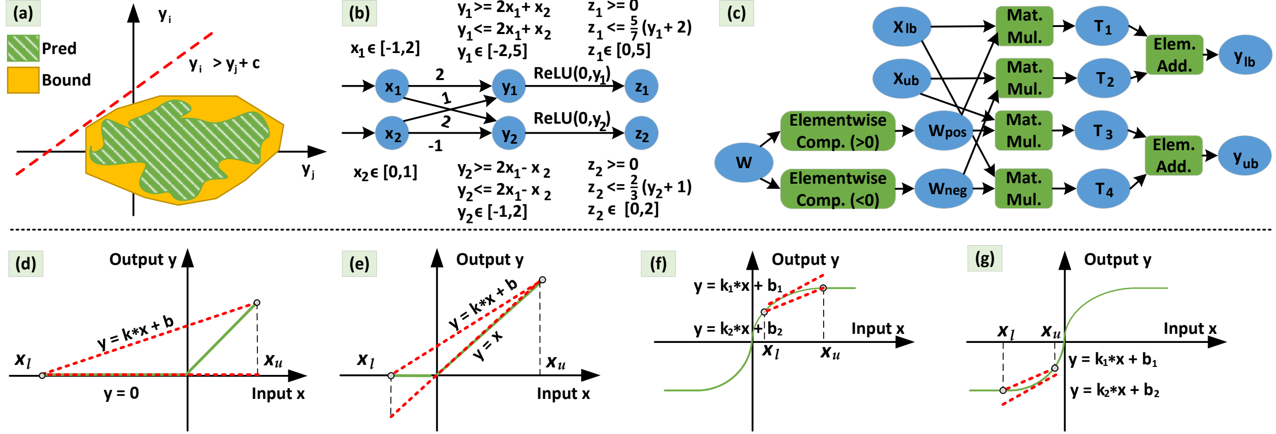


Figure 3: Illustration of transformer verification. (a) model prediction and verification bound; (b) an example of verifying a model with a fully connected layer and a ReLU layer; (c) computation graph of projection layer in transformer verification; (d)-(e) two types of bounds for ReLU layer; (f)-(g) two types of bounds for the *Tanh* layer.

and challenges for efficient transformer verification on GPUs (§2.3).

2.1 Transformer Verification

Standard Transformers. Transformer [8, 25, 38, 45] takes a sentence as input and predicts a label for this sentence (*e.g.*, hate speech or benign speech). Given a sentence with *Length* tokens, we usually first map each token to a pretrained embedding [28] of dimension *Dim_in* and represent the feature of a sentence as a tensor of shape $Length \times Dim_in$. For a batch of sentences, we have input feature X as a tensor of shape $Batch_size \times Length \times Dim_in$, where *Batch_size* is the number of sentences in a batch. Since the number of tokens varies across sentences, *Length* is set to the maximal number of tokens over all sentences in a batch.

A transformer has three types of operators. The first type is the *elementwise operator* that applies computation on individual feature scalars. For example, on each scalar x in the input feature, we have $ReLU(x) = max(0,x)$ and $Tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. The second type is the *matrix multiplication operator* that takes an input tensor X, a weight matrix W, and generates an output tensor $Y = XW$. We note that these two types are similar to operators in prior neural networks. The third type is the *dot product operator*, which is the key idea behind the transformer model. Informally speaking, it takes two input tensors Q and K of the same shape $Batch_size \times Length \times Dim_in$. Then, it computes an output tensor $Y = Q^T K$ of shape $Batch_size \times Length \times Length$ to measure the pairwise similarity between individual words in a sentence. This similarity can significantly improve the learning capacity of the model and the prediction accuracy.

Adversarial Attack on Transformers. Adversarial attack [2, 3, 15, 16, 17, 22] identifies small perturbations to input data X that can change the transformer prediction. Formally, consider a transformer $f(\cdot)$, an input sentence X, and a tolerable input perturbation bound ε, where the transformer correctly classifies X as a label i (*e.g.*, hate speech). In other words, the sentence has label i and $y_i > y_j$ for any $j \neq i$ where y_i is the predicted probability. Adversarial attack identifies a slightly perturbed sentence $X' = X + \eta$ such that $\eta \in B(0,\varepsilon)$ and there exists a label j (*e.g.*, benign speech) such that $y_i < y_j$. This perturbed sentence X' is an *adversarial example*.

Transformer Verification. Transformer verification [4, 18, 35, 42] computes a maximum bound ε and mathematically proves that there does not exist an adversarial example X' within the ε-ball of X (*i.e.*, $(X' - X) \in B(0,\varepsilon)$). Verifying transformers is challenging since transformers are essentially non-convex functions. The key idea of transformer verification is to utilize linear bounds as an approximation to NN predictions. We illustrate transformer verification at the model prediction layer in Fig. 3(a). Given these linear bounds, transformer verification can simply check if the predictions insides the bounds satisfy certain linear requirements, such as $y_i > y_j + c$, where c is a positive number. As illustrated in Fig. 3(a), this bound-based approach is sound since the linear bound covers the non-convex area of NN predictions.

We show an example of bound-centric computation of transformer verification in Fig. 3(b). Consider a fully connected layer $Y[j] = \sum_{i=1}^{n} W[j,i] \cdot X[i]$ where $Y[j]$, $W[j,i]$, and $X[i]$ are scalars. Here, we skip the index for batch size and length for notation simplicity. A formal summary of notations can be found in Table 1. For each neuron $X[i]$, there is a lower and a upper bound

$$X[i] \geq X_{lb}[i] + X_{lw}[i] * \vec{\varepsilon}, \quad X[i] \leq X_{ub}[i] + X_{uw}[i] * \vec{\varepsilon}$$

where $X_{lb}[i]$ and $X_{ub}[i]$ are scalars, $X_{lw}[i]$, $X_{uw}[i]$, and $\vec{\varepsilon}$ are vectors. For the input neurons, we have $X_{lb}[i] = X_{ub}[i] = X[i]$, $X_{lw}[i]$ and $X_{uw}[i]$ are one-hot vectors with 1 at the index i and 0 at other indices. Given this linear bound, we can compute *concretized* bounds for each neuron as

$$X_l[i] = X_{lb}[i] - \varepsilon * ||X_{lw}[i]||, \quad X_u[i] = X_{ub}[i] + \varepsilon * ||X_{uw}[i]|| \qquad (1)$$

where $||\cdot||$ computes the norm with reduction operations.

Table 1: Notations in transformer verification.

W	Transformer weights. Shape: $Dim_in \times Dim_out$
X	Input feature tensor. Shape: $Batch_size \times Length \times Dim_in$
X_{lb}, X_{ub}	The tensor of lower and upper bound bias of input features. Shape: $Batch_size \times Length \times Dim_in$
X_{lw}, X_{uw}	The tensor of lower and upper bound weights of input features. Shape: $Batch_size \times Length \times Dim_in \times Dim_out$
X_l, X_u	The tensor of concretized lower and upper bounds of input features. Shape: $Batch_size \times Length \times Dim_in$

When computing the bounds for output neuron $Y[j]$, we note that bound computation depends on the sign of weights $W[j,i]$. In particular, we have upper bounds $Y_{ub}[j]$ as

$$\begin{aligned} Y[j] \leq & Y_{ub}[j] + Y_{uw}[j] * \vec{\varepsilon} \\ = & (\sum_{W[j,i]\geq 0} W[j,i] \cdot X_{ub}[i] + \sum_{W[j,i]<0} W[j,i] \cdot X_{lb}[i]) \\ & + (\sum_{W[j,i]\geq 0} W[j,i] \cdot X_{uw}[i] + \sum_{W[j,i]<0} W[j,i] \cdot X_{lw}[i])) * \vec{\varepsilon} \end{aligned} \quad (2)$$

The lower bounds can be computed in a similar way. This bound computation (Eq. 2) is significantly different from standard NN computation since it explicitly considers the sign of weights. Previous transformer verification directly exploits the standard DL frameworks to build a computation graph (Fig. 3(c)) for computing bounds, which leads to inefficient memory access and computation overhead. We will discuss the opportunities and challenges of efficient transformer verification in §2.3.

For the same NN layer, diverse bound computation designs may still be developed to provide tighter bounds on NN predictions. We illustrate two types of bounds for the ReLU layer in §2(d)-(e) and two types of bounds for the Tanh layer in §2(f)-(g). A tighter bound (*i.e.*, less space between linear bounds and ReLU function) is preferred to provide a better linear bound approximation to NN prediction. For example, consider the concretized lower bound $X_l[i]$ and upper bound $X_u[i]$ for an input neuron $X[i]$, when we have $abs(X_l[i]) > abs(X_u[i])$, linear bound in Fig. 3(d) is preferred over the linear bound in Fig. 3(e) since the former one provides a tighter approximation. This diversity in bound design adds more complexity to developing frameworks for transformer verification.

2.2 Deep Learning Frameworks on GPUs

GPUs have been widely exploited to accelerate deep learning workload [13, 39, 40, 46, 49]. Efficiently mapping deep learning workloads to the GPU computing and memory hierarchy is usually the key to improve performance [11, 23, 41, 50, 51]. GPU computing hierarchy contains threads, warps, and blocks [29]. Each block has multiple warps and each warp has exactly 32 threads that compute with single-instruction-multiple-data (SIMD). GPU memory can be generally treated as a hierarchy of registers, shared memory, and global memory. Accessing registers is much faster than accessing shared memory, which is faster than accessing global memory. Each thread can only access its own registers and threads in a block cannot access shared memory from other blocks.

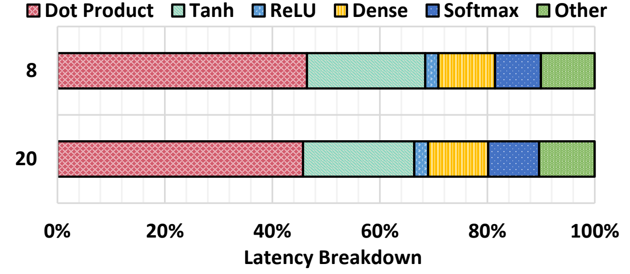


Figure 4: Latency breakdown of transformer verification on sentences with length 8 and 20. Here, we show the latency of verifying individual operators such as dot product and Tanh.

Many DL frameworks [6, 30, 54] have been developed recently to efficiently support NN workload on GPUs. Early works such as PyTorch [30] take user-specified computation graphs for neural networks and maps towards hand-tuned kernels on backend platforms (*e.g.*, GPUs). However, this approach usually builds upon kernels developed for standard NNs and cannot efficiently support transformer verification computation. Recent works, such as TVM [6] and Ansor [54], can automatically generate such backend kernels based on a set of heuristic rules on fusion and operator optimizations. However, these heuristic rules are developed specifically for standard NNs. Naively incorporating these rules into transformer verification may lead to unsatisfactory performance due to the significant difference in computation patterns. For example, Fig. 3(c) shows the computation graph for utilizing the kernels of standard NNs on transformer verification. This approach leads to heavy sparsity and redundant memory access. In particular, only half of the elements in W_{pos} and W_{neg} are non-zero values, leading to 50% sparsity. To this end, we build Faith, the first framework for efficient transformer verification on GPUs.

2.3 Opportunities and Challenges

In this section, we introduce optimization opportunities and challenges in enabling efficient transformer verification.

We show the latency of verifying individual transformer operators in Fig. 4. We profile this latency breakdown based on the state-of-the-art transformer verification implemented with PyTorch [30]. We have three major observations. First, dot product accounts for around 45% latency. Dot product takes two input tensors Q and K where both inputs may be perturbed during adversarial attack, which is significantly different from matrix multiplication that only one input (*i.e.*, feature X) may be perturbed. This adds complexity to the verification of dot product operators [35] and longer latency. Second, elementwise operators such as Tanh and ReLU account for a large portion of latency in transformer verification. This is significantly different from standard NNs where elementwise operators can usually be fused with remaining operators and show low latency. Third, we observe that ma-

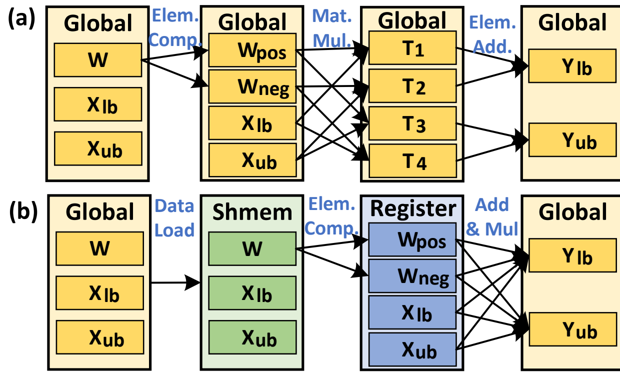


Figure 5: Illustration of Semantic-aware Kernel Fusion. We show the memory access pattern before and after applying semantic-aware kernel fusion in (a) and (b), respectively.

trix multiplication and softmax accounts for certain latency.

Opportunities: There are two major opportunities to accelerate transformer verification. The first opportunity is to exploit the semantics of transformer verification to minimize redundant memory access and computation. Our investigation shows that transformer verification has rich semantic information (*e.g.*, 50% sparsity in W_{pos} and W_{neg}), which can be exploited to accelerate transformer verification. The second opportunity is to exploit the modern GPU architectures to efficiently support diverse computing patterns in transformer verification. One example is to accelerate abundant reduction computation in Eq. 1.

Challenges: Although these ideas sound promising, the efforts to realize the benefits are non-trivial due to several challenges. First, transformer verification shows significantly different computing patterns from standard NNs. Straightforwardly borrowing optimizations for standard NNs such as kernel fusion can hardly bring similar benefits. Second, while exploiting GPU architecture supports may bring benefits, we still need specialized designs as a synergy between architecture and specialized computing patterns. Moreover, exploiting advanced GPU architecture supports will add more complexity to the search space of optimized kernels which motivates novel autotuning optimizations.

3 Semantic-aware Computation Graph Transformation

In this section, we propose *semantic-aware computation graph transformation* for efficient transformer verification. We first propose **semantic-aware kernel fusion** to fuse kernels within a transformer layer. It contains two novel types of fusions – *weight-paring based fusion* and *double bound based fusion*. Then, we propose **bound-aware cross-layer fusion** to efficiently fuse kernels across transformer layers.

3.1 Semantic-aware Kernel Fusion

The semantic-aware kernel fusion fuses operators in a single transformer layer to minimize memory access. Different from standard transformers, a single layer in transformer verification usually involves multiple kernels to compute the bounds adaptively to the sign of weights, as discussed in §2.1. Existing transformer verification [35, 42] usually uses a set of GPU kernels developed for standard transformers to serve the need for transformer verification. We illustrate the memory access pattern of this baseline approach in Fig. 5(a). These kernels need to independently read data from the global memory of GPUs and lead to heavy memory overhead. Moreover, these kernels fail to exploit semantic information in transformer verification and show heavy redundancy during memory access. For example, baseline approaches usually first split the weight matrix W into two weight matrices W_{pos} and W_{neg} according to weight signs and then use each matrix for computing lower and upper bounds. Here, these two split matrices W_{pos} and W_{neg} have the same shape of $M \times N$ as the weight matrix W. However, reading these matrices independently requires loading $2MN$ scalars, which leads to redundant memory access.

We propose semantic-aware kernel fusion to minimize such memory overhead by exploiting transformer verification semantics and GPU memory hierarchies (*i.e.*, global memory, shared memory, and registers). We illustrate our semantic-aware kernel fusion in Fig. 5(b). Our key insight is to first load data collaboratively from global memory and only distinguish data semantics (*e.g.*, W_{pos} and W_{neg}) at the register level to mitigate redundant memory access. In particular, we identify *weight-paring based fusion* and *double bound based fusion* as the two most important semantics in transformer verification.

Weight-pairing based fusion. We first propose weight-paring-based fusion to mitigate redundant memory access when reading W_{pos} and W_{neg}. Our key observation is that the zero values in W_{pos} are exactly the position of non-zero values in W_{neg}. Formally, we have $W_{pos} + W_{neg} = W$. To this end, instead of using an operator to split weight matrix W into W_{pos} and W_{neg}, we first load the matrix W from global memory to shared memory without distinguishing the sign of individual scalars. Then, we split the weight matrix W into W_{pos} and W_{neg} when loading data from shared memory to registers, as illustrated in Fig. 5(b). In our design, we only need to load MN scalars from global memory, which leads to significantly reduced memory access compared with loading $2MN$ scalars in baseline approaches.

Double bound based fusion. Our second optimization is a double-bound-based fusion. One important semantics in transformer verification is to multiply the same weight matrix with lower and upper input bounds (*e.g.*, X_{lb} and X_{ub}) to compute the output bounds (*e.g.*, Y_{lb} and Y_{ub} in Fig. 5(b)). Meanwhile, when computing the bound for output neurons, we usually need to read both lower and upper bounds for

computation. For example, when computing the upper bound of output neurons, we need to read upper bound when weight is positive and read lower bound when weight is negative. Suppose the input bounds X_{lb} and X_{ub} have shape $N \times K$, we need to load $4NK$ scalars during transformer verification.

Instead, we propose to fuse the computation of lower and upper bounds such that the lower and upper bounds only need to be loaded once to save memory access. In particular, we first use threads across GPU blocks to collaboratively load tiles of input matrices from global memory to shared memory, which can be accessed by different GPU threads. Here, we use shared memory to enable data sharing across GPU threads since different threads may multiply the same input bound scalar with different weight scalars (*e.g.*, multiplying the first row in X_{lb} and X_{ub} with various columns in W). Then, each thread loads independent data from shared memory to registers and directly accumulates output bounds Y_{lb} and Y_{ub} in registers. We note that this design further improves performance by eliminating the redundant global memory access during generating Y_{lb} and Y_{ub}.

3.2 Bound-aware Cross-layer Kernel Fusion

Bound-aware cross-layer kernel fusion fuses the verification of kernels across multiple transformer layers to further minimize memory access. Existing frameworks for accelerating standard NNs usually rely on a set of rules to fuse kernels. One popular example is to fuse convolution kernel with the following elementwise kernels (*e.g.*, ReLU kernel for elementwise comparison with 0). However, these rules usually cannot be applied to fuse kernels for transformer verification. For example, verifying the ReLU kernel requires first a concretization operation with a global reduction to compute the concretized bounds for a neuron and then applies different computation according to the concretized bounds (see §2.1).

To this end, we propose a set of rules for cross-layer kernel fusion in transformer verification. In particular, we recognize three types of operators. The first type is *input-reduction-compute* that conducts reduction or concretization operation on the input data before computation. One example is verifying nonlinear activation functions such as *ReLU* and *Tanh* that requires concretized bounds to apply different computation. Another example is the *softmax* operator that computes a global summation for normalization. The second type is *strict-elementwise* that contains only elementwise computation and does not require concretization or global summation. The third type is *dense-computation* such as matrix-matrix multiplication kernels. In our cross-layer kernel fusion design, we can always fuse a *dense* operator with its following *strict-elementwise* operator. However, we cannot fuse *dense* operator with *input-reduction-compute* due to the concretization or reduction operation. In addition, we can fuse *input-reduction-compute* with its following *strict-elementwise* operator. Finally, we can fuse multiple *strict-elementwise* operators (*e.g.*, elementwise addition and multiplication).

4 Verification-specialized Kernel Crafter

In this section, we propose a verification-specialized kernel crafter to efficiently map transformer verification towards modern GPUs. We exploit intrinsic properties (*e.g.*, abundant reduction operations) of transformer verification which are significantly different from standard transformer operators. One major challenge in building the kernel crafter is the large diversity in verification designs across operators (see Fig. 3(d)-(g)). To tackle this challenge, we first propose a *verification pattern categorization* to abstract such diversity and provide a small set of computing patterns over verification of diverse operators. Then, we propose three optimizations to efficiently support these computing patterns of transformer verification.

4.1 Verification Pattern Categorization

While there are diverse bound designs across different operators, we characterize transformer verification into four typical computing patterns. Based on this characterization, Faith can abstract the diversity in bound designs into a combination of computing patterns and exploit optimizations towards individual computing patterns for improving performance. Similar to standard NNs, one important computing pattern is *generalized matrix multiplication (GEMM)* when verifying projection layers and fully connected layers. Matrix multiplication is the major bottleneck in standard NNs and has been well-optimized by existing DL frameworks. Besides GEMM, transformer verification introduces three other time-consuming computing patterns, which are highlighted as follows:

The first computing pattern is *generalized vector reduction*. One typical source of generalized vector reduction is concretization that computes the norm and generates the concretized lower and upper bounds for individual neurons (see Eq. 1). Formally, consider a matrix $X = [\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m] \in \mathbb{R}^{m \times n}$ where $\vec{x}_i = [x_{i,1}, x_{i,2}, \cdots, x_{i,n}]$ are vectors of length n. The generalized vector reduction computes an output $Y = [y_1, y_2, \cdots, y_n] \in \mathbb{R}^n$ that satisfies

$$y_i = reduction(\vec{x}_i) = \sum_{j=1}^{n} f(x_{i,j}), \;\; i \in \{1, 2, \cdots, m\} \qquad (3)$$

Here, $f(x)$ is an elementwise function that takes a scalar input and generates a scalar output. One example for $f(x)$ is x^2 when computing the L_2 norm for input vectors.

The second computing pattern is *generalized elementwise multiplication* which appears frequently when verifying elementwise operators such as ReLU and Tanh. Formally, consider a concretized lower bound $l \in \mathbb{R}^{m \times n}$ and an upper bound $u \in \mathbb{R}^{m \times n}$ where $l_{i,j}$ and $u_{i,j}$ are concretized lower and upper bounds for the neuron at position (i, j). Let $X \in \mathbb{R}^{m \times n}$ be the input values. The generalized elementwise multiplication

computes an output $Y \in \mathbb{R}^{m\times n}$ that satisfies

$$y_{i,j} = f(l_{i,j}, u_{i,j}) * x_{i,j}, \quad i \in \{1,2,\cdots m\}, j \in \{1,2,\cdots,n\} \quad (4)$$

Here, transformer verification introduces a function $f(\cdot,\cdot)$ that takes the lower and upper bounds for an input neuron and computes a scaling parameter which is multiplied with the input value of this neuron. One example is the tangent line between the concretized lower and upper bounds when verifying Tanh layer, which accounts for more than 20% latency as we profiled in Fig. 4. Another example is $f(l_{i,j}, u_{i,j}) = 1$ when verifying ReLU layer and $l_{i,j}$ is non-negative. While $f(\cdot,\cdot)$ shows large diversity across operators, we stress that the same computing pattern is shared across these operators such that a uniform framework can be applied to improve performance.

The third computing pattern is *generalized scalar-vector multiplication*. This computing pattern exists widely when verifying dot products in the self-attention layer of transformers. This computing pattern accounts for more than 40% latency in transformer verification, as discussed in Fig. 4. Formally, consider a vector $S = [s_1, s_2, \cdots, s_m] \in \mathbb{R}^m$ and a matrix $X = [\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m] \in \mathbb{R}^{m\times n}$, where s_i are scalars and $\vec{x}_i = [x_{i,1}, x_{i,2}, \cdots, x_{i,n}]$ are vectors of length n. The generalized scalar-vector multiplication computes an output $Y = [\vec{y}_1, \vec{y}_2, \cdots, \vec{y}_n] \in \mathbb{R}^{n\times n}$ that satisfies

$$\vec{y}_i = f(s_i) * \vec{x}_i = [f(s_i) * x_{i,1}, f(s_i) * x_{i,2}, \cdots, f(s_i) * x_{i,n}], \quad i \in \{1,2,\cdots,m\} \quad (5)$$

Here, $f(\cdot)$ is a function that takes a scalar input and generates a scalar output.

Generability to diverse NN operators. Faith can effectively support verifying diverse NN operators such as SiLU and Leaky ReLU. Our key insight is that verifying diverse NN operators usually share the same generalized computing pattern while the concrete computation formula might be different. For example, $SiLU(x) = \frac{x}{1+e^{-x}}$ is an activation function that has significantly different concrete computation formula from $ReLU(x) = max(0,x)$. However, both verifying SiLU and ReLU can be treated as the generalized elementwise multiplication (Eq. 4) and the same optimizations can be applied to improve performance.

In the following sections, we first demonstrate a *workload-adaptive reduction* to improve the performance of generalized vector reduction (Eq. 3). We then propose a *sharing-oriented workload scheduling* to improve the performance of generalized elementwise multiplication (Eq. 4). Finally, we demonstrate *broadcast-aware super threading* to efficiently support the generalized scalar-vector multiplication (Eq. 5).

4.2 Workload-adaptive Reduction

Transformer verification contains abundant reduction operations where a sequence of scalars are summed up into one scalar. One common reduction operation is the concretization operation that computes the concretized lower and upper bounds for individual neurons, as discussed in §2. Another common reduction operation is the softmax operation that is applied in each self-attention layer for measuring the relationship between individual words. These reduction operations pose challenges between parallelism and data locality. One baseline approach is to use a single thread to read and accumulate a sequence of scalars as illustrated in Fig. 6(a). However, this approach usually leads to low parallelism and fails to exploit abundant threads in GPUs. For example, we need 32 iterations to accumulate 32 scalars. Another baseline approach is to first split this sequence of scalars into multiple chunks and allocate one thread to each chunk for accumulation. Then, each thread writes the accumulated results for each chunk to global memory and uses an additional thread to finally accumulate the sum of each chunk. While this approach improves parallelism, it requires expensive global memory access and high overhead.

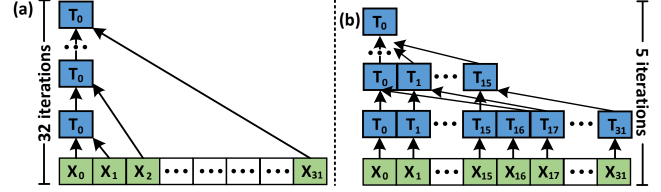


Figure 6: Illustration of Workload-adaptive Reduction. (a) Sequential Mode; (b) Parallel Mode. Here, x_i and T_i are the i-th data and thread, respectively.

Workload-adaptive Reduction with length $n = 32$. We propose a *workload-adaptive reduction* to fully exploit GPU memory hierarchies and the inter-register communication functionalities. We illustrate our design in Fig. 6(b). Our design achieves high parallelism by enabling multiple threads for reduction simultaneously. Meanwhile, we avoid the expensive data communication through global memory and exploit only efficient registers. In particular, we use 32 threads (*i.e.*, a warp) to read these 32 scalars simultaneously from global memory. Considering these 32 scalars are consecutive in global memory, we can efficiently load them with 32 threads through coalesced memory access. Then, we exploit the specialized instruction `_shfl_down_sync` to directly communicate data in registers across individual threads. As illustrated in the parallel mode of Fig. 6(b), our design involves only five iterations of cross-thread data communication to generate the final accumulated result, rather than the 32 iterations in the sequential mode of Fig. 6(a).

Workload-adaptive Reduction with Arbitrary Length n. For an arbitrary length n, one naive approach is to repeatedly use 32 threads to reduce 32 scalars and then use 1 thread to accumulate the final results. However, this approach may lead to unnecessary communication across threads. Suppose we are accumulating a vector of length $n = 32k$, we need 5 iterations for reducing every 32 scalars, leading to $5k$ iterations in total for accumulating the vector. Instead, we propose

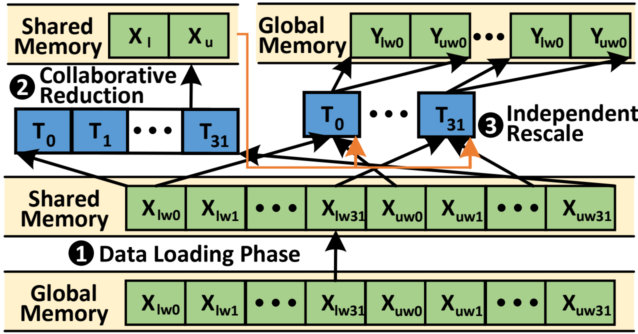


Figure 7: Illustration of sharing-oriented workload scheduling

a *hybrid mode* to minimize the number of iterations while still achieving high parallelism. In particular, we first split the input sequence into chunks where each chunk contains 32 scalars. Then, we use 32 threads to read one chunk simultaneously from global memory and accumulate individual chunks iteratively. For example, the 1-st thread accumulates the 1-st scalar in each chunk. Here, the accumulation is conducted in registers and does not require communication across threads. Finally, we apply a single 5-iteration reduction across 32 threads. In total, our design has only $k+5$ iterations which are significantly less than $6k$ iterations in the naive approach.

4.3 Sharing-oriented Workload Scheduling

We propose *sharing-oriented workload scheduling* to efficiently verify elementwise operators. Different from standard transformers, verifying elementwise operators, especially nonlinear ones (*e.g.*, ReLU and Tanh), accounts for a large portion of latency in transformer verification as we discussed in Fig. 4. Verifying these operators usually first requires computing a concretized lower bound X_l and upper bound X_u for each input neuron and then computes the bounds for the output neuron. Different signs of concretized input bounds usually lead to different computations for output bounds, which could easily lead to warp divergence and unsatisfactory performance. Moreover, when computing the output bound weights (*i.e.*, Y_{lw} and Y_{uw}) for a neuron, we need to repeatedly use the same input bounds which leads to extra memory overhead.

To efficiently verify elementwise operators, we propose *sharing oriented workload scheduling* to minimize memory access and improve performance. Our key observation is that the same set of input bound weights X_{lw} and X_{uw} are used to compute the concretized input bounds X_l and X_u, while these input weights are also used for computing the output bound weights Y_{lw} and Y_{uw}. Instead of repeatedly loading X_{lw} and X_{uw}, we can exploit the GPU memory hierarchies to cache X_{lw} and X_{uw} and minimize the global memory access to improve the overall performance.

As illustrated in Fig. 7, we use a set of T(=32) threads to first (Step ❶) load input bound weights X_{lw} and X_{uw} from global memory to shared memory. Here, T is a hyper-parameter to balance the parallelism and compute intensity, which will be selected in §5. Then (Step ❷), these T threads load input bound weights from shared memory and collaboratively compute the concretized lower and upper bounds X_l and X_u, following our design in §4.2. These concretized lower and upper bounds are stored in shared memory which can be accessed by individual threads. Finally (Step ❸), each thread independently loads individual X_{lw} and X_{uw} scalars from shared memory and rescales according to the concretized bounds X_l and X_u. Here, all threads in a warp are computing the output bound weights for the same neuron and the concretized input bounds are the same across threads in a warp. Thus, all threads in a warp can apply the same rescaling computation and avoid warp divergence. We also note that input bound weights are only loaded once from global memory which mitigates redundant global memory access.

4.4 Broadcast-aware Super Threading

We propose *broadcast-aware super threading* to efficiently support generalized scalar-vector multiplication, as discussed in Eq. 5. One naive approach is to use one thread to read a scalar s_i and a vector $\vec{x}_i$ and computes the generalized scalar vector multiplication $f(s_i)\vec{x}_i$. However, this approach fails to exploit the parallelism opportunities in generalized scalar vector multiplication. Another approach is to split the vector $\vec{x}_i$ into multiple chunks and use one thread for each chunk. However, this approach requires threads to repeatedly read the same scalar s_i from global memory and shows redundant memory access.

Instead, we propose a broadcast-aware super threading to achieve high parallelism while minimizing memory access. We consider two types of super threading for generalized scalar vector multiplication. The first type is a group of 32 threads (*i.e.*, a warp for one vector). When using 32 threads to compute the multiplication between a scalar s_i and a vector $\vec{x}_i$, these 32 threads can read the scalar s_i once, broadcast across threads with modern GPU memory, and compute $f(s_i)$ simultaneously. Based on this broadcast, we can mitigate the redundant memory access that each thread repeatedly read the same scalar s_i. The second type is a group of $32t$ threads (*i.e.*, t warps for one vector). In this case, we use one warp to read the scalar s_i and use shared memory to broadcast s_i across warps.

5 Expert-guided Autotuning Optimization

Considering the large design space of optimization towards GPUs, one natural question arises: *Can we effectively incorporate hardware knowledge to find optimal operator implementation?*

Existing works such as TVM [6] and Ansor [54] usually autotune operator implementations in a hardware-agnostic way. In particular, these works extract implementation-specific parameters such as tiling size and use a cost model to implicitly

learn the relationship between these parameters and performance. However, there are two drawbacks in this hardware-agnostic approach. First, there is a complex interaction between implementation and the hardware properties, which could be hard to be implicitly learned by the cost model. For example, existing works [12, 20, 24, 43] on hand-tuning large matrix-matrix multiplication operators usually maximize the number of registers in use to improve cache performance. However, this optimization is also limited by the number of registers for each GPU thread since exceeding such limitation may lead to register spilling [27] and a significant performance drop. A careful reasoning on the interaction between the implementation-specific parameters (*e.g.*, the number of registers for caching data) and the hardware properties (*e.g.*, the number of registers per thread) is usually necessary to maximize the performance. To tackle this challenge, we propose an *expert-guided autotuning optimization* to automatically reason both implementation-specific parameters and hardware properties. In particular, we have the following designs.

Rule-based Expert Knowledge Metafile. We propose a *rule-based expert knowledge metafile* to capture hardware properties. This metafile only needs to be set once for each type of GPUs and requires limited manual efforts. In particular, we consider two types of rules. The first type is *hard rules* which represents hardware limitation such as the maximal shared memory size and the maximal number of registers per thread. Violating these rules may lead to significant performance drop such as register spilling. The second type is *soft rules* which represents intrinsic trade-offs related to the hardware properties such as the number of streaming multiprocessors (SM) and the number of threads per SM. One typical design choice is the number of threads per block which will be mapped to threads on the same SM. Allocating more threads per block usually leads to better parallelism for the sub-task assigned to a block. However, allocating more threads per block may also hinder executing multiple blocks on the same GPU SM hardware and lead to worse overall parallelism.

Expert-guided Cost Model. We propose an *expert-guided cost model* to automatically tackle the complex interaction between implementation-specific parameters and hardware properties. Given a set of candidate operator implementations, we have two phases to select the optimal implementation. The first phase is to estimate the shared memory and register usage for each candidate. We rule out candidates that consume more shared memory and registers than hardware capacity, as specified in the expert knowledge metafile.

The second phase is to train a cost model for the remaining candidates and use the cost model to select the best candidate. We use XGBoost [5] as the cost model. It takes as input the implementation-specific parameters (e.g., tiling size) for candidates and the hardware properties (e.g., shared memory size). We use the cost model to predict the latency of candidates and select top-k candidates with low latency. Finally, we profile the latency of these top-k candidates on GPUs and use

Dataset	#Train	#Val	#Test	Length		
				min	mean	max
SST	67,349	872	1,821	4	25	62
YELP	560,000	0	38,000	5	98	128

Table 2: Dataset statistics

the profiled latency to further fintune our cost model. We repeat this procedure for a pre-defined iterations (=5 by default) and select the implementation with the lowest latency.

When training the cost model, we construct the training dataset as follows. We randomly select a small number of candidates and use their implementation-specific parameters and the hardware properties as feature X. Then, we profile the latency of each candidate implementation on GPUs as the label Y. We collect these (X,Y) as the training dataset to train the cost model.

6 Evaluation

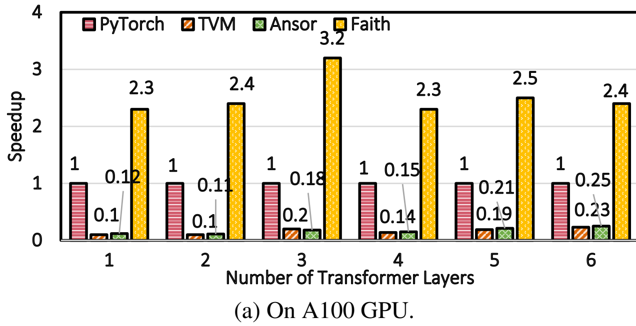
In this section, we comprehensively evaluate Faith over various datasets and GPU backends. We first present our experiment setup in §6.1. Then, we show the overall speedup on end-to-end transformer verification in §6.2. Finally, we provide more optimization analysis on individual transformer layers in §6.3.

6.1 Experiment Setup

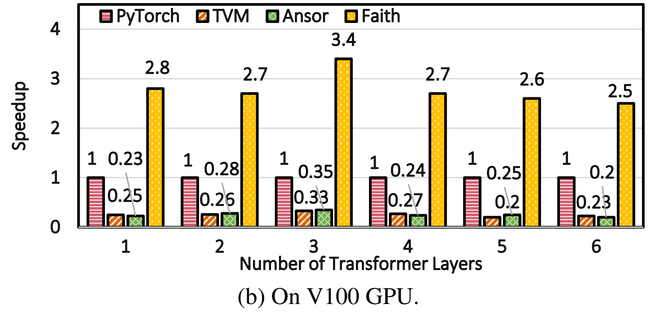
Baselines. We compare Faith with the state-of-the-art transformer verification [35] based on PyTorch. We further compare with TVM [6] and Ansor [54], as stronger baselines. TVM and Ansor are two state-of-the-art deep learning compilers for standard neural networks. We feed the pytorch model into TVM and Ansor through relay frontend [34] which will automatically optimize transformer verification performance. While TVM and Ansor take minutes to compile an operator implementation, we do not incorporate this compilation latency and record only inference latency for a fair comparison.

Datasets. We evaluate two popular datasets, Yelp [53] and SST [36], following the setting in state-of-the-art transformer verification [35]. These two datasets are widely used in the natural language processing for analyzing sentiment in languages. We summarize the statistics of these two datasets in Table 2. SST dataset contains 67,349 training sentences, 872 validation sentences, and 1,821 testing sentences. In SST dataset, there are 4 to 62 tokens in each sentence and the average number of tokens in a sentence is 25. YELP dataset contains 560,000 sentences as training data and 38,000 sentences as testing data. In YELP dataset, there are 5 to 128 tokens in each sentence and the average number of tokens in a sentence is 98.

Transformer Networks. We evaluate Faith on transformer networks with 1 to 6 layers to demonstrate the performance on large models. Following popular transformer settings, each transformer layer has 4 attention heads and an embedding size

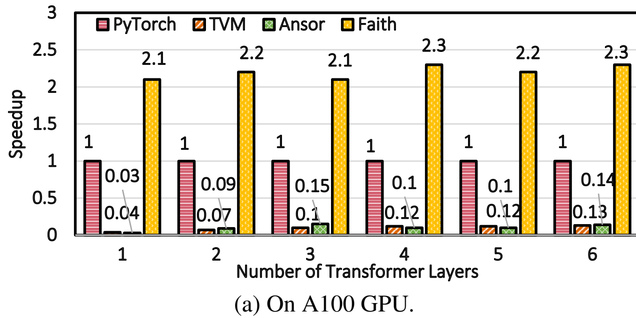


(a) On A100 GPU.

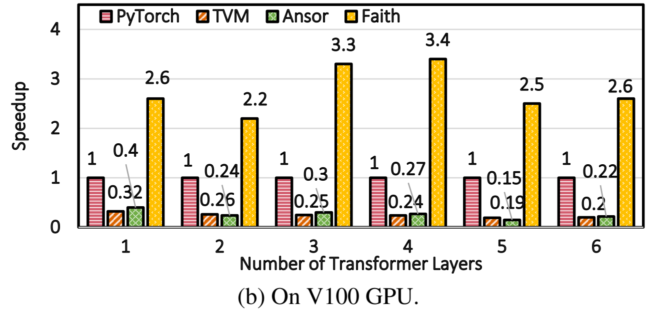


(b) On V100 GPU.

Figure 8: Overall speedup on SST dataset.



(a) On A100 GPU.



(b) On V100 GPU.

Figure 9: Overall speedup on Yelp dataset.

of 128. Furthermore, we study the Faith performance under diverse embedding sizes in §6.3.

Experiment Configuration. We evaluate with an NVIDIA A100 GPU and an NVIDIA V100 GPU to show Faith performance on various GPU backends. The host server with A100 GPUs is an AMD EPYC 7742 64-Core Processor and runs Ubuntu 20.04 with CUDA 11.3. The host server with V100 GPUs has 32 cores of Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and runs Ubuntu 16.04 with CUDA 10.1.

6.2 Overall Performance

We show the overall speedup on SST dataset and Yelp dataset in Fig. 8 and Fig. 9, respectively. We show the performance improvement over transformers with diverse numbers of layers from 1 to 6, which covers popular settings in the natural language processing domain. While the length of input sentences may have an impact on the performance improvement, we show the averaged speedup over all testing sentences in this section and study the impact of sentence length in §6.3. We compare Faith with the PyTorch baseline following existing transformer verification open-source implementations [35]. We further compare Faith with two state-of-the-art deep learning frameworks (*i.e.*, TVM and Ansor) to provide a comprehensive comparison, as we discussed in §6.1.

We show the overall speedup on SST dataset and A100 GPU in Fig. 8(a). Compared with PyTorch, we observe 2.3× to 3.2× speedup (2.5× on average). We contribute this performance improvement to our semantic-aware computation graph transformation (§3) and verification-specialized kernel crafter (§4). We further observe 17.2× and 15.9× speedup over TVM and Ansor, respectively. The main reason is that TVM and Ansor focus on optimizing standard neural networks and fail to efficiently support verification-specific computing patterns, as discussed in §2.2. While Faith and these three baselines show different performance, we stress that the same verification bounds are generated, and the only difference resides in system optimizations. Comparing across different numbers of transformer layers from 1 to 6, the performance improvement remains similar around 2.5×. This result shows that Faith can efficiently support transformer verification with diverse numbers of transformer layers. We show the overall speedup on SST dataset and V100 GPU in Fig. 8(b). We have similar observation about the results on A100 GPU which shows that Faith can effectively adapt to diverse GPU backends, thanks to expert-guided autotuning optimization (§5).

We show overall speedup on Yelp dataset and A100 GPU in Fig. 9(a). Sentences in YELP dataset has 5 to 128 tokens (98 on average), which is longer than sentences in SST dataset with 4 to 62 tokens (25 on average). This provides an opportunity to show Faith performance on long sentences. Overall, we observe 2.1× to 2.3× speedup (2.2× on average) when comparing with the PyTorch baseline. We also observe 26.7× and 28.3× speedup on average over TVM and Ansor, respectively. This speedup is similar to the performance improvement on SST dataset and shows the good generality of Faith over diverse input data. We also have similar observations on Yelp dataset and V100 GPU in Fig. 9(b).

6.3 Optimization Analysis

In this section, we show speedup from individual Faith optimizations. We first show speedup on *verification of matrix*

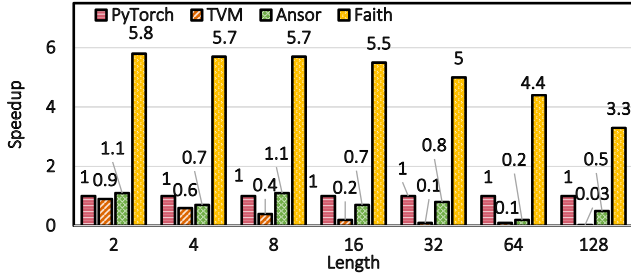


Figure 10: Speedup on verification of matrix multiplication over the diverse lengths. Embedding Size: 128.

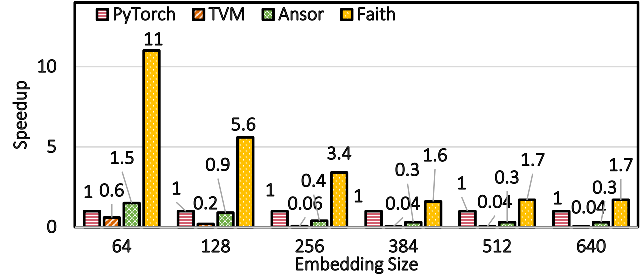


Figure 11: Speedup on verification of matrix multiplication over the diverse embedding sizes. Length: 16.

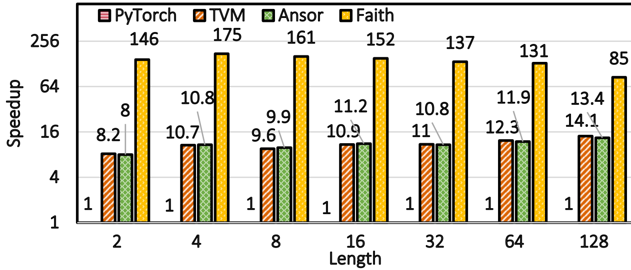


Figure 12: Speedup on verification of ReLU over the diverse lengths. Embedding Size: 128.

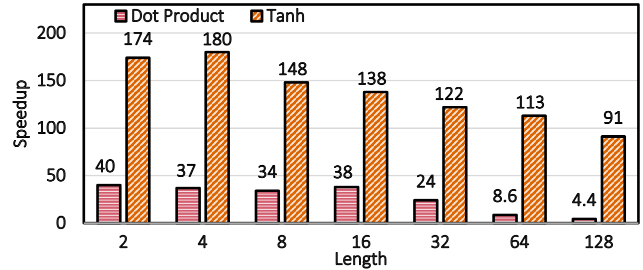


Figure 13: Speedup on verification of Tanh and dot product over the diverse lengths. Embedding Size: 128.

multiplication over the diverse lengths and diverse embedding sizes. Verification of matrix multiplication plays an important role in verifying projection layers and fully connected layers in transformers. Then, we show the benefits on verification of ReLU, verification of dot product, and verification of Tanh, which in total accounts for around 70% latency in transformer verification. Since we observe similar performance on A100 GPU and V100 GPU, we focus on A100 GPU and omit results on V100 GPU in this section due to page limits.

Performance benefits on verification of matrix multiplication. We show speedup on verification of matrix multiplication over the diverse lengths in Fig. 10. We study the speedup over diverse lengths from 2 to 128, following the setting in the popular natural language processing datasets as summarized in Table 2. Overall, we observe 5.1× speedup on average over the PyTorch baseline. This result shows significant performance benefits from utilizing Faith on accelerating transformer verification. Comparing across lengths, we observe a higher speedup of 5.54× over the PyTorch baseline on shorter sentences with 2 to 32 words. The reason is that our autotuning optimization (§5) automatically adjusts the number of threads and memory layout to improve the parallelism. We achieve a smaller speedup of 3.85× on longer sentences with 64 and 128 words. For these longer sentences, we have achieved high occupancy on GPUs and the speedup is limited by the hardware capability.

Surprisingly, we observe that TVM and Ansor achieve 0.33× and 0.73× speedup, which is significantly slower than PyTorch baselines on verification of matrix multiplication. The main reason is that TVM and Ansor focus on accelerating standard NNs and cannot efficiently support computing patterns in the verification of matrix multiplication (Fig. 3(c)). Instead, Faith exploits a semantic-aware kernel fusion (§3.1) to efficiently support such computing patterns in verification.

We show speedup on verification of matrix multiplication over the diverse embedding sizes in Fig. 11. We study embedding size from 64 to 640 following popular transformer settings. We note that transformer in natural language processing usually adopts a relatively small embedding size (*e.g.*, 64 to 256), which is different from convolutional neural networks in computer vision that adopts a large embedding size (*e.g.*, 1024). Overall, Faith achieves 4.2× speedup on average over the PyTorch baseline. This result shows that Faith can improve performance over diverse embedding sizes. We also observe that Faith achieves larger speedup for smaller embedding sizes, which is similar to the case when verifying matrix multiplication over diverse lengths.

Performance benefits on verification of ReLU. We show speedup on verification of ReLU over diverse lengths in Fig. 12. As we discussed earlier in §4.1, verification of ReLU represents an important computing pattern of verifying elementwise operators. Due to similar behaviors between diverse lengths and embedding sizes, we focus on verification over diverse lengths and keep embedding size as 128, which is a popular setting in transformers. Overall, Faith achieves 141× speedup over PyTorch baseline. This large speedup shows it promising to accelerate verification of elementwise operators. Besides, Faith achieves 13.4× and 13.5× speedup over TVM and Ansor. The reason is that our *workload-adaptive reduction* (§4.2) can significantly improve parallelism during reduction and *sharing-oriented workload sharing* can minimize memory access with GPU memory hierarchy.

#Layers	1	2	3	4	5	6
PyTorch	9.1	18	25	28	31	37
Faith	4	7.2	7.8	10.9	12.6	15.4

Table 3: Latency on SST dataset and A100. Unit: Second.

Performance benefits on verifying Tanh and dot product layers. We show the speedup from Faith over the PyTorch baseline on verification of Tanh and verification of dot product in Fig. 13. We skip the results of TVM and Ansor since these two frameworks do not support computing patterns in verification of Tanh and verification of dot product. Here, we show results of verification of Tanh since it is a popular elementwise operator in transformer verification. We also show results of verification of dot product since it accounts for around 45% latency in transformer verification. Overall, we observe that Faith achieves 138× speedup on average for verification of Tanh. This result is similar to the performance improvement for verification of ReLU, since both Tanh and ReLU are elementwise operators and share benefits from the same set of optimizations. We also observe that Faith achieves 26.5× speedup on average for verification of dot product. This result shows the performance benefits from semantic-aware kernel fusion (§3.1) and broadcast-aware super threading (§4.4) that mitigate redundant memory access.

Raw latency on transformer verification. We show the raw latency for transformer verification on the SST dataset and NVIDIA A100 GPU in Table 3. Faith requires only a few seconds to verify the NN prediction on a long sentence (with on average 25 tokens). More specifically, when verifying transformers with 1 to 6 layers, Faith only requires 4 to 15.4 seconds to verifying a sentence. This results brings transformer verification to the level of being practical for use.

7 Discussion

Why Faith performs better than prior approaches. Existing frameworks, such as PyTorch, TVM, and Ansor, only support limited computation patterns for standard NNs. They cannot directly support bound-centric computation patterns in transformer verification. While several frameworks like TVM allow autotuning for diverse operators, there is no magic. They still rely on hand-written GPU kernels (e.g., matrix multiplication) as the parametric templates (e.g., with tiling size as a parameter) and can only tune these tiling sizes. When applying to bound-centric computation patterns, they will break an operator for transformer verification into several hand-written GPU kernels for standard NNs. This leads to significantly higher memory access when aggregating computation results across GPU kernels into one transformer verification output.

Instead, Faith provides direct support for bound-centric computation patterns. Instead of breaking into several GPU kernels for standard NNs, we consider the bound-centric computation patterns as a whole and design a set of optimizations to reduce the memory and computation cost. For example, we found the lower and upper bounds are usually multiplied with the same weight matrix and can be loaded once to reduce memory overhead.

Practicality of transformer verification with Faith. Faith brings transformer verification to be practical by consuming only around 10 seconds to verify a long sentence (e.g., 25 tokens). We remark that transformer verification is one of the hottest topics in deep learning. Hundreds of related papers have been published in top deep learning conferences. The performance is essential to bring transformer verification into practical applications. However, existing efforts mainly reside in the algorithmic domain. In this paper, we build the first framework for efficient transformer verification on GPUs. Our work will open a new system research direction on developing high-performance systems for deep learning verification.

8 Conclusion

Verifying the robustness of transformers draws increasing attention from both the academic and industry fields over the recent years. Unfortunately, an efficient system design for transformer verification is still yet to come. Existing transformer verification still exploits standard neural network frameworks which are unoptimized towards transformer verification workload. The main reason is that efficient systems for transformer verification require both expertise from the machine learning community on mathematical verification designs and the system community on efficient memory and parallelism designs.

In this paper, we propose a Faith framework for efficient transformer verification. Specifically, we first design a set of semantic-aware computation graph transformations to fully exploit fusion opportunities in transformer verification at the computation graph level. Then, we propose a verifier-specialized kernel crafter to efficiently map fused verification kernels towards modern GPUs with minimized memory overhead and improved parallelism. Finally, we propose an expert-guided autotuning to dynamically optimize kernels according to the transformer verification workload and GPU backend characteristics. Comprehensive experimental evaluation shows that Faith significantly improves the performance of transformer verification over state-of-the-art frameworks.

Looking ahead, we believe our work in transformer verification would highlight a new direction on developing high-performance systems for deep learning verification. This will encourage system experts with diverse backgrounds to build the next-generation deep learning systems and facilitate the wide application of secure deep learning.

References

[1] Nader Akoury, Kalpesh Krishna, and Mohit Iyyer. Syntactically supervised transformers for faster neural machine translation. In *ACL (1)*, pages 1269–1281. Association for Computational Linguistics, 2019.

[2] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani B. Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *EMNLP*, pages 2890–2896. Association for Computational Linguistics, 2018.

[3] Melika Behjati, Seyed-Mohsen Moosavi-Dezfooli, Mahdieh Soleymani Baghshah, and Pascal Frossard. Universal adversarial attacks on text classifiers. In *ICASSP*, pages 7345–7349. IEEE, 2019.

[4] Gregory Bonaert, Dimitar I. Dimitrov, Maximilian Baader, and Martin T. Vechev. Fast and precise certification of transformers. In *PLDI*, pages 466–481. ACM, 2021.

[5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594. USENIX Association, 2018.

[7] Junyan Cheng, Iordanis Fostiropoulos, Barry W. Boehm, and Mohammad Soleymani. Multimodal phased transformer for sentiment analysis. In *EMNLP (1)*, pages 2447–2458. Association for Computational Linguistics, 2021.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics, 2019.

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

[10] Facebook. How facebook uses super-efficient ai models to detect hate speech. `https://ai.facebook.com/blog/how-facebook-uses-super\/-efficient-ai-models-to-detect-hate-speech/`.

[11] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient GPU serving system for transformer models. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 389–402. ACM, 2021.

[12] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *PPoPP*, pages 278–291. ACM, 2021.

[13] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. APNN-TC: accelerating arbitrary precision neural networks on ampere GPU tensor cores. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, pages 37:1–37:13. ACM, 2021.

[14] Siddhant Garg, Thuy Vu, and Alessandro Moschitti. Tanda: Transfer and adapt pre-trained transformer models for answer sentence selection. `https://www.amazon.science/publications/tanda-transfer-and-adapt-pre-trained-\/transformer-models-for-answer-\/sentence-selection`.

[15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR (Poster)*, 2015.

[16] Yu-Lun Hsieh, Minhao Cheng, Da-Cheng Juan, Wei Wei, Wen-Lian Hsu, and Cho-Jui Hsieh. On the robustness of self-attentive models. In *ACL (1)*, pages 1520–1529. Association for Computational Linguistics, 2019.

[17] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. Is BERT really robust? A strong baseline for natural language attack on text classification and entailment. In *AAAI*, pages 8018–8025. AAAI Press, 2020.

[18] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[19] Bumsoo Kim, Junhyun Lee, Jaewoo Kang, Eun-Sol Kim, and Hyunwoo J. Kim. HOTR: end-to-end human-object interaction detection with transformers. In *CVPR*, pages 74–83. Computer Vision Foundation / IEEE, 2021.

[20] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of SGEMM on fermi and kepler gpus. In *CGO*, pages 4:1–4:10. IEEE Computer Society, 2013.

[21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *ICLR*. OpenReview.net, 2021.

[22] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. In *NDSS*. The Internet Society, 2019.

[23] Xin-Chun Li, De-Chuan Zhan, Jia-Qi Yang, and Yi Shi. Deep multiple instance selection. *Sci. China Inf. Sci.*, 64(3), 2021.

[24] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A coordinated tiling and batching framework for efficient GEMM on gpus. In *PPoPP*, pages 229–241. ACM, 2019.

[25] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[26] Yu Lu, Jiali Zeng, Jiajun Zhang, Shuangzhi Wu, and Mu Li. Attention calibration for transformer in neural machine translation. In *ACL/IJCNLP (1)*, pages 1288–1298. Association for Computational Linguistics, 2021.

[27] Paulius Micikevicius. Local memory and register spilling. https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

[28] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[29] Nvidia. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[31] Lihua Qian, Hao Zhou, Yu Bao, Mingxuan Wang, Lin Qiu, Weinan Zhang, Yong Yu, and Lei Li. Glancing transformer for non-autoregressive neural machine translation. In *ACL/IJCNLP (1)*, pages 1993–2003. Association for Computational Linguistics, 2021.

[32] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[34] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new IR for machine learning frameworks. In *MAPL@PLDI*, pages 58–68. ACM, 2018.

[35] Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. Robustness verification for transformers. In *ICLR*. OpenReview.net, 2020.

[36] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, pages 1631–1642. ACL, 2013.

[37] Hao Tang, Donghong Ji, Chenliang Li, and Qiji Zhou. Dependency graph enhanced dual-transformer structure for aspect-based sentiment classification. In *ACL*, pages 6578–6588. Association for Computational Linguistics, 2020.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[39] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In Andreas Krall and Thomas R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 41–53. ACM, 2018.

[40] Yuke Wang, Boyuan Feng, and Yufei Ding. QGTC: accelerating quantized graph neural networks via GPU tensor core. In Jaejin Lee, Kunal Agrawal, and Michael F.

Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 107–119. ACM, 2022.

[41] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for GNN acceleration on gpus. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 515–531. USENIX Association, 2021.

[42] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In *NeurIPS*, 2020.

[43] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing batched winograd convolution on gpus. In *PPoPP*, pages 32–44. ACM, 2020.

[44] Fuzhi Yang, Huan Yang, Jianlong Fu, Hongtao Lu, and Baining Guo. Learning texture transformer network for image super-resolution. In *CVPR*, pages 5790–5799. Computer Vision Foundation / IEEE, 2020.

[45] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*, pages 5754–5764, 2019.

[46] Han-Jia Ye, Yi Shi, and De-Chuan Zhan. Identifying ambiguous similarity conditions via semantic matching. In *CVPR*. Computer Vision Foundation / IEEE, 2022.

[47] Linwei Ye, Mrigank Rochan, Zhi Liu, and Yang Wang. Cross-modal self-attention network for referring image segmentation. In *CVPR*, pages 10502–10511, 2019.

[48] Da Yin, Tao Meng, and Kai-Wei Chang. Sentibert: A transferable transformer-based architecture for compositional sentiment semantics. In *ACL*, pages 3695–3706. Association for Computational Linguistics, 2020.

[49] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. G-TADOC: enabling efficient gpu-based text analytics without decompression. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1679–1690. IEEE, 2021.

[50] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Poclib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE Trans. Parallel Distributed Syst.*, 33(2):459–475, 2022.

[51] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. TADOC: text analytics directly on compression. *VLDB J.*, 30(2):163–188, 2021.

[52] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *NeurIPS*, pages 4944–4953, 2018.

[53] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS*, pages 649–657, 2015.

[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, pages 863–879. USENIX Association, 2020.

[55] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable DETR: deformable transformers for end-to-end object detection. In *ICLR*. OpenReview.net, 2021.