Empowering In-Browser Deep Learning Inference on Edge Devices with Just-in-Time Kernel Optimizations

Fucheng Jia*
Central South University
Microsoft Research
fuchengjia@csu.edu.cn

Shiqi Jiang
Microsoft Research
shijiang@microsoft.com

Ting Cao†
Microsoft Research
ting.cao@microsoft.com

Wei Cui
Microsoft Research
weicu@microsoft.com

Tianrui Xia*
University of Southern California
Microsoft Research
tianruix@usc.edu

Xu Cao
Microsoft Research
caox@microsoft.com

Yuanchun Li
Institute for AI Industry Research (AIR), Tsinghua University
liyuanchun@air.tsinghua.edu.cn

Qipeng Wang*
Peking University
Microsoft Research
wangqipeng@stu.pku.edu.cn

Deyu Zhang
Central South University
zdy876@csu.edu.cn

Ju Ren
Tsinghua University
renju@tsinghua.edu.cn

Yunxin Liu
Institute for AI Industry Research (AIR), Tsinghua University
liuyunxin@air.tsinghua.edu.cn

Lili Qiu
Microsoft Research
liliqiu@microsoft.com

Mao Yang
Microsoft Research
maoyang@microsoft.com

ABSTRACT

Web is increasingly becoming the primary platform to deliver AI services onto edge devices, making in-browser deep learning (DL) inference more prominent. Nevertheless, the heterogeneity of edge devices, combined with the underdeveloped state of Web hardware acceleration practices, hinders current in-browser inference from achieving its full performance potential on target devices.

To address this issue, this paper presents the pioneering in-browser inference system, nnJIT, which enables just-in-time (JIT) auto-generation of optimized computing kernels for edge devices. nnJIT is built upon two novel techniques that significantly reduce kernel search and compilation overhead while improving performance firmly: Tensor-Web Compiling Co-Design lowers compiling costs by around 100× through eliminating redundant and ineffective compiling passes; Web-Specific Lite Kernel Optimization Space reduces kernel tuning costs by focusing on Web programming requirements and efficient device resource utilization, pruning the optimization space from millions to only dozens.

nnJIT[1] is evaluated for modern models, *e.g.,* BART, T5, and Llama 2, on a range of edge devices including laptops and smartphones using different browsers and hardware from ARM, Intel, AMD and Nvidia. The results show that nnJIT can achieve up to 8.2× faster within 30 seconds compared to the existing baselines.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**.

KEYWORDS

In-Browser Deep Learning Inference, Just-in-Time Kernel Optimizations, WebAssembly, WebGPU

ACM Reference Format:
Fucheng Jia, Shiqi Jiang, Ting Cao, Wei Cui, Tianrui Xia, Xu Cao, Yuanchun Li, Qipeng Wang, Deyu Zhang, Ju Ren, Yunxin Liu, Lili Qiu, and Mao Yang. 2024. Empowering In-Browser Deep Learning Inference on Edge Devices with Just-in-Time Kernel Optimizations. In *The 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '24), June 3-7, 2024, Tokyo, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3498361.3538932

*Research interns at Microsoft Research.
†Corresponding author.

MobiSys '24, June 3-7, 2024, Tokyo, Japan

ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3538932

[1]Code: https://aka.ms/nnjit-web

1 INTRODUCTION

Web applications that run through a browser are increasingly popular on edge devices, thanks to their notable benefits such as cross-platform compatibility, effortless *click-and-run* deployment, ease of maintenance, and seamless integration between edge and cloud services [28].

Presently, there is a trend towards integrating Deep Neural Network (DNN) services directly into Web applications, enabling in-browser inference. Systems that facilitate in-browser inference have been developed, such as ONNX Runtime Web[45], TensorFlow.js[39], WebDNN[4], and brain.js[1]. In-browser inference can provide a more responsive user experience and enhanced privacy protection by avoiding round-trips to cloud, as well as save the expense of cloud computing resources. In-browser inference is made viable by continuous advances in Web programming techniques, such as WebAssembly (abbr. Wasm) [3] and WebGPU [6], coupled with the ever-increasing processing power of edge devices.

However, current in-browser inference systems suffer from two major drawbacks. Firstly, their predefined kernels do not account for device diversity. Unlike cloud, edge devices are hetergenous, equipped with a range of CPUs, GPUs, memories, browsers, OS [37], and may also experience interference from other running applications. The *one-for-all* approach delivers poor performance across devices. As we will show in Fig. 2, a predefined kernel can be several times slower than our device-customized kernels. Secondly, these systems lag behind cutting-edge Web programming techniques *i.e.,* WebGPU[13]. Their handwritten kernels for each new Web programming backend (*e.g.,* JavaScript, Wasm, WebGL[5]) necessitate significant rewriting efforts.

To address these challenges, kernel auto-generation techniques such as TVM[12], Ansor[46] and FlexTensor [47] can be employed to automatically generate customized kernels without manual efforts. However, these techniques require *ahead-of-time* kernel generation for known hardware. Given a tensor computation, these techniques search for the potential optimal implementation from many possible implementations (e.g., different loop order). For each selected implementation candidate, they generate and compile the kernel code to evaluate on the target devices. This *select-generate-evaluate* process will be repeated until the specified end condition is met. To approach the optimal, this searching process can take hours even days to run, and also requires on-device kernel evaluation. This process is suitable for cloud with known and finite hardware. Unfortunately, Web applications are intentionally designed to function on diverse edge devices. Considering the huge number of different devices, generating kernels *ahead-of-time* for each device is impractical. Consequently, achieving optimal in-browser inference performance for each edge device remains an unresolved challenge.

To tackle it, we rethink the specialties of Web. Compared to native precompiled inference systems, in-browser inference offers the distinct advantage of *online kernel updating*. Furthermore, in-browser inference typically runs repeatedly over an duration, such as for video and document processing. This distinctive feature provides the opportunity and time budget for just-in-time (JIT) kernel customization after encountering the actual edge device.

Based on this insight, we present nnJIT, the first in-browser DNN inference system with the unique ability to JIT auto-generate and continuously improve optimized kernels during inference for target edge devices, leading to a gradual speedup towards optimal performance. Both CPU and GPU are supported through generating kernels in the state-of-the-art (SOTA) Web programming interfaces respectively, *i.e.,* Wasm for CPU and WebGPU for GPU.

To realize this system, the main challenge lies in enabling JIT generation of optimized kernels, a feat that has never been accomplished before. This is due to the huge time cost of current optimized kernel generation, stemming from (1) the compiling cost and (2) the vast kernel optimization space. Tensor computations, implemented as multi-level loops, create a vast kernel optimization space due to variations in loop arrangements like tiling sizes. To find the optimized kernel, the kernel tuning process iteratively selects potential candidate from this space for compilation and evaluation on the target device. The compiling for each candidate can take minutes to complete numerous transforming passes. Previous works [10, 29, 48] try to reduce the space size or improve the searching method. However, the remaining space is still too large to enable JIT kernel optimization, or necessitates known hardware to build performance model ahead-of-time.

By comparison, nnJIT can facilitate JIT generation of optimized kernels for diverse edge devices, based on our key findings of Web programming. (1) Web programming interface is designed with simple instruction sets and execution model for efficiency and security, which does not require complex compiling optimizations. Moreover, mostly compiling optimizations for Web programming interface are overlapped with kernel optimization space, *e.g.,* loop unrolling, rendering them unnecessary. (2) Strict Web requirements for security and portability convey consistent performance pattern across devices, *e.g.,* costly memory allocation. This consistency removes the need for related candidates in the kernel optimization space to be evaluated on target devices.

Based on the two findings, we propose two novel techniques accordingly. The first is *Tensor-Web compiling co-design*. Taking Wasm compilation as an example. Rather than the separated tensor-level and language-level (*i.e.,* Wasm) compiling, nnJIT employs a unified compiling pipeline that directly compiles tensor computation to in-browser executable, completely eliminating the costly invocation of separated language-level compiler *e.g.,* LLVM [2] or Emscripten [17]. The unified compiling pipeline provides the capability of co-designing the tensor and language compiling optimizations to avoid redundant and ineffective ones. This new compiling pipeline dramatically reduces the cost per candidate, from minutes to milliseconds.

The second technique is *Web-specific lite kernel optimization space*. The space is designed by offline consistent primitive setting decided by Web requirements, and online inconsistent primitive setting decided by target edge device. As Web requirements cause consistent performance patterns across devices, to identify their impact, we compose a microbenchmark suite that traverses the tensor compiling primitives (*i.e.,* code transformations conducted on tensor) such as loop order and unroll, in a *one-variable-at-a-time* manner. The suite is evaluated offline to identify the efficient primitive settings. Besides the consistent ones, there are primitives *inconsistent* across devices depending on hardware specifications. The dominant is tiling size that mostly impact the hardware utilization. A proper tile size can balance the contention between parallel

hardware execution and advanced memory accesses. We use formulated kernel hardware usage and heuristics to select promising tile sizes to construct the lite kernel optimization space. Consequently, the number of candidates in space is reduced, from millions to only dozens.

Based on the two techniques, we develop the NNJIT. After the initial model and kernels are downloaded onto the target edge device, NNJIT generates the lite kernel optimization space. Candidates in the space are compiled one-by-one using our unified compiling pipeline and evaluated on the device, interleaved with the inference process with limited overhead. Better kernels are continuously replaced online, gradually approaching the optimal. Considering the large number of clients on Web, candidate evaluation results and generated kernels are also crowdsourced from ones with similar device, achieving optimal kernels much faster.

We implement NNJIT on both Wasm for CPUs and WebGPU for GPUs. It can run on devices with browsers installed that support Wasm or WebGPU. Wasm is supported by mainstream browsers. WebGPU, although still in its early stages, shows great promise. Thanks to our JIT kernel auto-generation, NNJIT is the first to supports WebGPU for complex models, serving as a strong showcase for our advantages.

NNJIT is evaluated on representative modern models, with suitable size for edge devices, including T5 [35], BART [27], GPT-2 [34], RoBERTa [30] and Llama 2 7B [41]. Mainstream browsers are used *i.e.,* Chrome, Microsoft Edge, Firefox and Opera, which together take 87% of market share [36]. We evaluate on a range of smartphones, laptops and desktops, *e.g.,* Pixel 4, Vivo X30, SurfaceBook 3, Lenovo V9000, MagicBook and HP EliteDesk, equipped with ARM CPU (Cortex-A76, A78), Intel CPU (I9 12900H), AMD CPU (Ryzen 5800H), Intel GPU (HD 630), AMD GPU (Radeon), and Nvidia GPUs (RTX 3050, 3000, 3070Ti). The results show within 30 seconds, NNJIT can achieve up-to 20.1× faster kernels, and 8.2× faster model inference compared to SOTA inference frameworks. To summarize, our main contributions include:

- This paper proposes the first in-browser inference system that enables JIT optimized kernel generation.
- The Tensor-Web compiling co-design avoids the ineffective and redundant optimizations, reducing the compiling cost from minutes to milliseconds.
- The Web-specific lite kernel space design is guided by both Web programming requirement and efficient utilization of hardware resource, reducing the optimization space from millions to dozens.
- The evaluation is done on modern transformer models and a range of edge devices, achieving up to 8.2× speedup compared to SOTA frameworks.

2 BACKGROUND AND MOTIVATION

2.1 DL Inference in Web Browsers

Compared to cloud- or 5G edge-based solutions for DNN inference, on-device inference provides specific advantages in reducing cloud operation costs and providing improved privacy. As reported by Microsoft and Google [20, 21], on-device inference has the potential to achieve zero cost for providing DNN services to a large number of customers. Additionally, as stated in [22], there is always a risk of

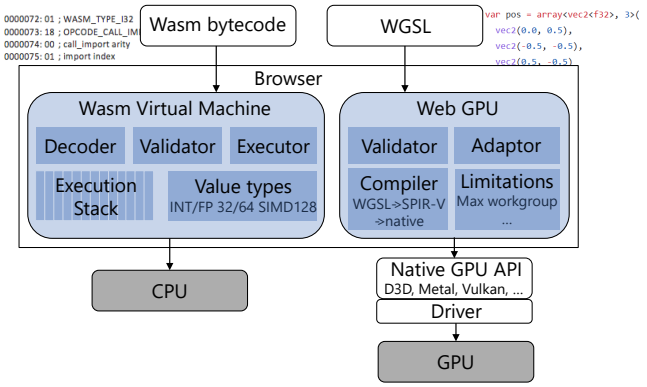


Figure 1: The Wasm and WebGPU support in browser.

data breaches when information travels over the internet. Therefore, even though network bandwidth is less of an issue these days, on-device inference attracts more and more attention compared to the cloud counterpart.

Nowadays, as many DNN models are directly integrated into Web applications [16], in-browser inference on device in gaining momenta [1, 4, 39, 45]. However, enabling DNN inference in modern Web browsers is nontrivial [32, 44]. Due to the security considerations, the sandbox mechanism is widely used within browsers, which isolates Web applications, scripts, and other contents from the underlying system. The sandbox environment prevents malicious code from accessing and modifying system resources and settings, meanwhile it also restricts the usage of the sophisticated native DNN inference libraries, such as Eigen [15] for CPU and cuBLAS [14] for GPU.

To make DL inference in browsers possible, alternative programming interfaces, hence backends, are proposed to use. JavaScript [25] is firstly leveraged to implement DL kernels and graphs in Web DL frameworks [39]. JavaScript has no-static data type and no vectorization support. Although some efforts like V8 Engine [18] could significantly accelerate JavaScript code, the DL execution with it is still extremely inefficient in JavaScript environment.

To cope with it, WebAssembly (Wasm) [3] is considered. Wasm is a compact binary format. Its runtime is a portable virtual machine running on the CPU. Fig. 1 shows the Wasm implementation in browsers. Wasm code is delivered in low-level bytecode, which can be decoded and executed more efficiently in the virtual machine. The bytecode needs to be validated for security. What's more, Wasm also takes advantage of advanced features of modern CPUs, *e.g.,* Single Instruction Multiple Data (SIMD). Therefore, it provides much better inference performance than JavaScript. Wasm is language-agnostic. High-level programming language like C and C++ could be compiled into Wasm bytecode.

GPUs are also utilized within browsers. For instance, WebGL has been integrated in TensorFlow.js, providing JavaScript interfaces to access GPU that originally enable rendering 3D graphics on Web pages. It is based on OpenGL ES 2.0 [19], a subset of OpenGL [33]. Thus, certain features are unavailable. Meanwhile as a rendering library, it failed to utilize computation pipelines in modern GPUs due to limited instructions.

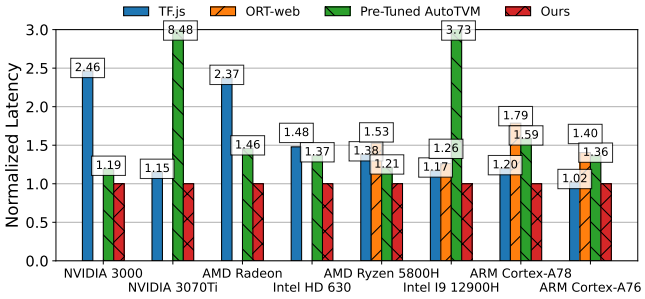


Figure 2: The normalized kernel latency of handwritten, pre-tuned, and our NNJIT for a MatMul ([M,K,N]=[640,768,2304]).

To unleash the power of GPU, WebGPU, the successor of WebGL, is proposed. WebGPU provides stronger computation ability, driving computation intensive DL kernels to execute more efficiently. WebGPU Shading Language (WGSL) is used to program. Fig. 1 shows the implementation of WebGPU in browsers. While running in browser, the WebGPU kernel is translated to native GPU APIs, such as Vulkan [42]. For portability, WebGPU also specifies limitations for the hardware usage. The validator is also introduced to check kernels for the security purpose.

Taking the advantages of the backends above, Web DL frameworks including TensorFlow.js (TF.js) and Onnx Runtime Web (Ort-Web), enable end-to-end in-browser inference for pretrained DL models. They all have relatively mature support for Wasm, and start to support WebGPU. The kernels shipped within these frameworks are usually handwritten or ported from native DL frameworks, *e.g.,* TensorFlow [38]. To optimize the kernels, kernel auto-generators such as TVM [12] are extended for Wasm and even WebGPU. However, generating kernels for Web usually takes extreme long time, *e.g.,* nearly 2 hours for one Matrix Multiplication (MatMul). Besides that, the performance of tuned kernels are almost far from the optimal, which we will discuss in the next.

2.2 Inference Performance Issues

To understand in depth the DL inference performance in browsers, we conduct the preliminary study, using a MatMul kernel to demonstrate. We have the following observations:

The one-for-all kernels are suboptimal across devices. Web applications are running on millions of edge devices equipped with diverse hardware. Different hardware prefers different kernel implementations. However, instead of designing customized kernels for each type of devices, at present the SOTA in-browsers inference frameworks deliver kernels in a one-for-all style. For instance, TF.js and ORT-Web ship handwritten kernels on Wasm and WebGPU. We execute the one-for-all MatMul kernels from TF.js, ORT-web (only support Wasm), and pre-tuned AutoTVM (without tuning on the target device) on AMD 5800H desktop CPU, ARM Cortex-A78/A76 mobile CPUs, Nvidia 3000/3070Ti GPU and Intel 630 GPU. The inference latency is illustrated in Fig. 2.

The results indicate the performance of pre-defined kernels is suboptimal compared to our device-customized kernels. Moreover, a single pre-defined kernel exhibits a wide range of performance gaps on different devices. For instance, the kernel from TF.js demonstrates a slowdown ranging from as little as 2% to as much as 246%

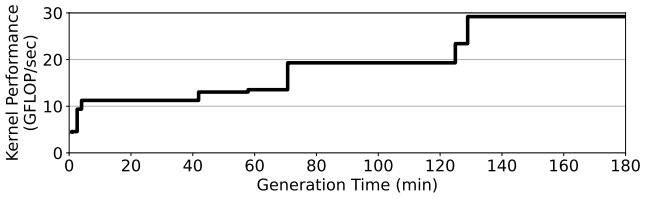


Figure 3: The generated MatMul kernel ([M,K,N]=[640,768,2304]) performance and generation time of TVM on AMD 5800H CPU.

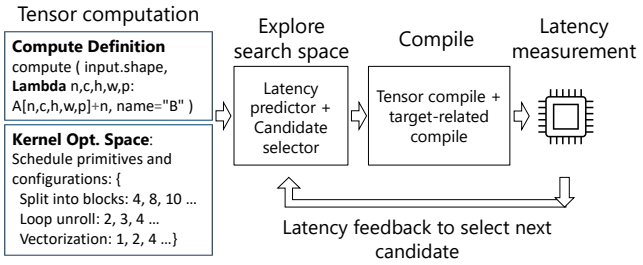


Figure 4: A common kernel generation pipeline.

when compared to customized ones. Similarly, without tuning, the generated kernel from TVM shows a slowdown of 19% to multiple times depending on devices. These results highlight the need for customized kernels tailored to each edge device.

The one-for-each kernels are currently impractical in Web scenarios. Based on the measurements presented above, one might consider generating kernels in a one-for-each style. However, this solution remains infeasible. We assessed the kernel generation time of TVM for a MatMul kernel on an AMD 5800H CPU device. It took nearly 2 hours to identify the kernel with the high performance (29.2 GFLOP/sec), with 437 tuning rounds. Typically, a deployed model contains several tens of kernels. Clearly, the one-for-each approach is impractical, particularly for Web scenarios where client diversity is substantial.

The prolonged kernel generation cost is due to two primary causes: the bloated compilation process and the exceedingly large kernel optimization space.

Fig. 4 illustrates a common kernel generation pipeline. The tensor computation is defined in a domain specific language. Its potential kernel implementations, which composes a kernel optimization space, are defined by *primitives* and the according configurations. A primitive is a kind of code transformation *e.g.,* loop unroll. A candidate from the kernel space can be described by a sequence of primitives and their configurations. The compiling process can then follow these primitives to conduct compiling IR (intermediate representation) transformations to generate the kernel. After that, the target language compiler *e.g.,* LLVM can be called to compile the kernel into executables for the target devices.

As the combination blowup of loop arrangement, the kernel optimization space is huge. Our analysis shows the size of a naive space for a MatMul (384×768×768) in WebGPU is around 42 M. Advanced searching algorithms and hardware performance models [10, 29, 43, 47, 48] are normally employed to only select promising candidates for compilation and evaluation on the target device.

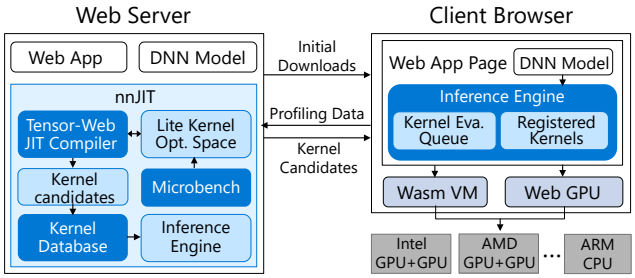


Figure 5: Overview of NNJIT.

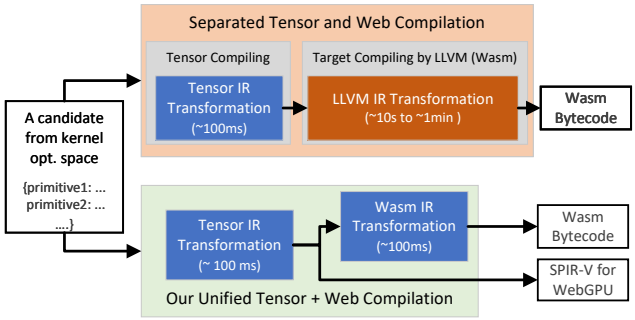


Figure 6: Our unified Tensor+Web compiling (lower half) compared to conventional separated Tensor and Web compiling (upper half).

Even so, thousands of candidates are generally evaluated before finding an optimized kernel implementation. The compiling cost for each candidate is around seconds to minutes depending on the kernel quality. The total generation cost of optimized kernel will be hours.

Clearly, to reduce the kernel generation cost for JIT, we need to reduce the compiling cost for each candidate, and reduce the number of candidates in the space. Therefore, we propose NNJIT. In the following sections, we will introduce the design principles and key techniques of NNJIT.

3 NNJIT OVERVIEW

Fig. 5 is the overview of NNJIT. It consists of four modules: the *tensor-web JIT compiler* for online kernel generation; the *inference engine* for executing inference tasks in the browser; the *micro benchmark suite* for offline exploration of the consistent primitive settings; and the *kernel database* for storing optimized kernels tailored to known devices. The whole kernel generation and inference process facilitated by NNJIT operates on both cloud and clients, as follows.

During the initialization phase, the browser on the client downloads the web page, the inference engine, the model and the initial kernels. The model encloses the weights and the optimized model graph (*e.g.,* operator fused) ready to deploy. The *inference engine* parses the model graph, registers the kernel for each operator to execute, as well as manages the memory usage. The initial kernels are determined by the server, using the client device indicator, *e.g.,* device name and ids. If the hardware on client have been explored, the optimal kernels would be used from the *kernel database* on server. Otherwise, the pre-defined and uncustomized ones are used meanwhile the JIT phase would be triggered.

During the JIT phase, the *tensor-web JIT compiler* on the server composes the lite kernel optimization space for each operator type. The compiler then subsequently generates the kernel for each candidate within the space. Between the server and the client, a kernel queue is established. Once a kernel is generated on server, it is pushed to the client via the queue. On client, the inference is executed repeatedly. Between every inference, the *inference engine* retrieves one kernel from the queue and measures its latency. Based on the measurement, the newly retrieved kernel might be re-registered if it is significantly faster than the current registered one, ensuring that the more efficient kernel is utilized in subsequent inferences. After testing all the kernels in the queue, the best kernel along with the measurement results are reported to the server. The server would update the *kernel database* according to the reports.

In accordance with our design, the *tensor-web JIT compiler* of NNJIT is lightweight and can be run either on the cloud or directly on clients. In our current implementation, we deploy it on the cloud, as this enables kernel reuse. Optimal kernels discovered by one device can be seamlessly shared with other devices possessing the same hardware through the cloud, thereby facilitating the concept of *crowdsourcing*.

The online kernel generation combined with JIT-styled inference ensures optimal performance on Web across edge devices. To facilitate this, we propose two key techniques that significantly reduce the kernel generation cost, *e.g.,* from hours to milliseconds for a single kernel. In the following sections, we will introduce these techniques in detail.

4 STREAMLINE COMPILATION PIPELINE THROUGH TENSOR-WEB CO-DESIGN

Each candidate in the kernel optimization space needs to be compiled and evaluated on the client device. Current compiling takes minutes to complete. Even the space only has dozens of candidates, the total compiling will take hours, not possible to support the online optimized kernel generation. To reduce the cost, this section introduces the possibilities of removing target-related compilation (Sec. 4.1), mapping directly from tensor-level IR to Wasm IR (Sec. 4.2), and only keep necessary optimization passes on Wasm IR (Sec. 4.3). Sec. 4.4 will briefly discuss compiling pipeline for WebGPU.

4.1 Unify Tensor-Web Compiling

Costly target-related compilation. As shown in Fig. 6, the conventional compiling process of tensor computation consists of two main separated steps: the tensor-level compilation and the followed target-related compilation (*e.g.,* Wasm). Generally, they are designed separately by different communities, each with their own specific purpose.

Tensor-level compilation transforms the tensor-level IR by the primitives and configurations of a picked candidate from the kernel optimization space, to generate a mapping of tensor computation to

a loop arrangement. This process is independent of the target execution environment. Target-related compilation, on the other hand, aims to generate the efficient executables on the target environment from any high-level programs. Therefore, after tensor-level compilation generates the loop, a separate target-related compilation library such as LLVM is normally invoked to generate the executables. As these target-related compilation libraries aim to compile any general-purpose programs, there are many compiling passes, taking long time to complete.

Specifically, for Wasm, the target execution environment is Wasm virtual machine running within browsers. The target-related compilation library is LLVM or Emscripten. The compilation by LLVM/Emscripten contributes the majority of total compilation cost, as shown in Fig. 6.

Feasibility of eliminating target-related compilation. We therefore explore the possibility to eliminate this target-related compilation, by identifying two opportunities.

Firstly, we could remove *ineffective optimizations*. From the target perspective, Wasm is designed with a simple expression-based instruction set and a stack-based execution model [23], for the purpose of easy decoding, running efficiency, and security. Consequently, many sophisticated compiling optimizations would be not effective or necessary, thus not needed, such as the ones for register allocation, instruction reordering, and memory disambiguation.

Secondly, we could remove *duplicated optimizations*. From the tensor perspective, the kernel optimization space which includes numerous possible kernel implementations, also encompasses many of the target-related compilation optimizations. For example, the unrolled loop generated by LLVM optimization pass is very likely included in the kernel optimization space, which will be evaluated as well. The separated tensor-level and Wasm-level compilation cannot avoid the redundancy. In addition, the tensor computation defined domain specific languages need no complex compiling optimizations used by general-purpose programs, such as the dead code elimination. Thus, it could be further streamlined.

Unified Tensor-Web compiling. The analysis above prompts us to redesign the compiling pipeline, which unifies the tensor-level and Wasm-level compilation as shown in Fig. 6. It removes the separated target-related compiling invocation, and compiles tensor computation directly to the target executables *e.g.*, Wasm bytecode. The tensor-level IR is directly mapped to Wasm IR, and then mapped to Wasm bytecode. As a premise, Wasm is designed to be the compiling target of any high-level languages, including C and C++. It can also be the target of tensor-level IR.

The optimization passes of different level IR's are co-designed, retaining only the necessary and non-repetitive ones. Through analyzing the generated code performance, we find almost all the optimization passes in LLVM can be covered in kernel optimization space. Only the ones closely related to Wasm instruction definition will be additionally needed to apply on the Wasm IR as the figure shows. These passes are very light weighted, taking about 100 ms to complete, tens or even hundreds of times less than calling LLVM.

4.2 Lower Tensor to Wasm

The primary challenge in directly lowering tensor IR to Wasm IR involves determining how to effectively map the statement-based high-level tensor IR to the expression- and stack-based low-level

```
@main = primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
preflattened_buffer_map = {A_1: A_3: Buffer(A_2, float32, [64, 64], []),
                           B_1: B_3: Buffer(B_2, float32, [64, 64], []),
                           C_1: C_3: Buffer(C_2, float32, [64, 64], [])} {
  ......
  for (m.outer: int32, 0, 16) {
    for (n.outer: int32, 0, 16) {
      ......
      for (k.outer: int32, 0, 2) {
        for (k.inner: int32, 0, 32) {
          ......
          C.local_1[0] = (C.local_1[0]
              + (broadcast(A[cse_var_3], 4)*B[ramp(cse_var_2, 1, 4)]))
          ......
```

(a) Tensor IR

```
(func $main (param $0 i32) (param $1 i32) (param $2 i32)
  ......
  (loop $label$1
    (loop $label$2
      ......
      (loop $label$3
        (loop $label$4
          ......
          (local.set $24
            ......
            (f32x4.mul
              (local.tee $38
                (v128.load32_splat offset=12288)(local.get $0)
              )
              (local.get $36)
            )
            ......
```

(b) WASM IR

Figure 7: Lower tensor IR to Wasm IR for MatMul.

Wasm IR. Fig. 7 uses a code snippet to illustrate the differences between the two IRs, by using a MatMul implementation as an example. Wasm IR has only been lowered from LLVM IR before. LLVM IR is also a lower-level IR than tensor IR. For example, LLVM IR has already lowered the high-level `for` statement in tensor IR.

As Fig. 7 shows, the tensor IR is represented as a sequence of statements, such as the `for` loop statement. Wasm IR, on the other hand, is composed of a sequence of expressions (enclosed by the parenthesis in the figure). Each expression is evaluated to produce a value. Wasm virtual machine to run Wasm bytecode is stack-based, in which instructions manipulate an implicit operand stack, popping argument values and pushing result values. This design is to fit the sandboxed and resource-limited environment of browsers.

The lowering of tensor IR to Wasm IR needs to consider both the expression and stack execution order.

Map `for` statement to an expression block. Algorithm 1 shows the transform of the `for` statement. We construct a nested sequence of expressions as a block enclosed by the Wasm `loop&end` instructions for this statement.

As shown in Algorithm 1, the sub-expressions, *e.g.*, loop variable calculation, are created while traversing the `for` node of the tensor IR (line 2-7). Then the expressions will be nested together as the execution order of the stack (line 8-10). During execution, the `br_if` will pop the condition result from the stack, and decide whether to branch to the loop label (line 5). The `loop` instruction introduces an implicit label, which serves as the target of the branch instruction. During the actual stack execution, the `loop` instruction pushes a new entry onto the control stack, and record the stack height. If the branch is taken, the stack pops up to the block's height before and proceed to the end of the block.

Algorithm 1: Lower Tensor IR to Wasm IR for loop

input : ForNode of Tensor IR $forNode$
output: LoopExpression of Wasm IR $loopExpr$

▷ for(loopVar=begin;loopVar<end;loopVar+=stride) body;
$loopVar \leftarrow$ createWasmVar();
$initLoopVarExpr \leftarrow$ makeLocalSet($loopVar$, $forNode$.begin);
$ltExpr \leftarrow$ makeBinary(Op::Lt, $loopVar$, $forNode$.end);
$brIfExpr \leftarrow$ makeBreak($loopVar$.label, $ltExpr$);
$addLoopVarExpr \leftarrow$ makeBinary(Op::Add, $loopVar$, $forNode$.stride);
$bodyExpr \leftarrow$ VisitStmt($forNode$.body);
$innerExpr \leftarrow$ makeBlock($bodyExpr$, $AddLoopVarExpr$, $brIfExpr$);
$loopExpr \leftarrow$ makeLoop($loopVar$.label, $innerExpr$);
$loopExpr \leftarrow$ makeBlock($initLoopVarExpr$, $loopExpr$);

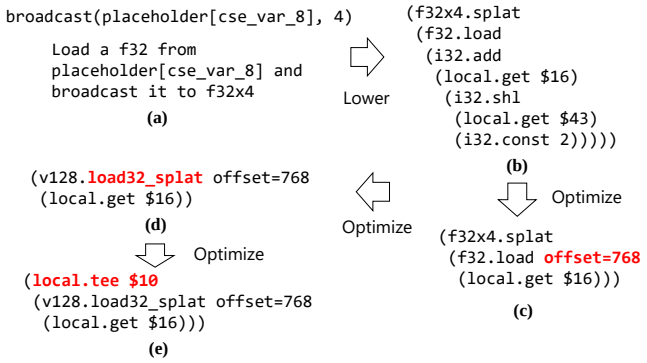


Figure 8: Wasm IR transformation by applying each of compiling passes: (a) tensor IR for a broadcast load statement; (b)lowered Wasm IR; (c) Wasm IR after offset load/store pass; (d) Wasm IR after combined instruction pass; (e) Wasm IR after load/store to variable pass.

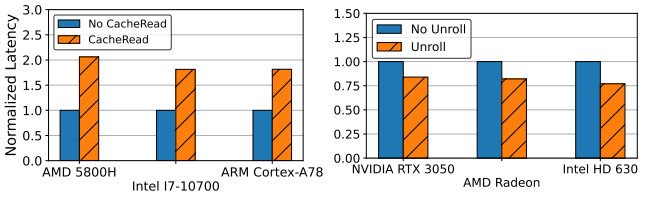
4.3 Compiling Optimizations for Wasm IR

As stated above, our compiling pipeline applies the optimization passes related to Wasm instruction definition to the Wasm IR. Only three passes are needed, as shown in Fig. 8: 1) offset load/store, 2) load/store to variable, and 3) combined instruction. Each pass is explained in detail below.

(1) Offset load/store pass is to eliminate constant address calculation for load/store instructions. Wasm code execution accesses a linear memory in the Wasm virtual machine. Wasm provides the offset augmented load/store instruction to avoid the address calculation. This pass is to utilize this instruction. By applying it as shown in Fig. 8 (b, c), five additional instructions can be eliminated for each load/store. This optimization can speed up generated kernels by 2.7×.

(2) Combined instruction pass is to eliminate separated instructions if possible. It is to apply the combined instruction, *i.e.,* `v128.load_splat`, provided by Wasm. This `load_splat` combines `load` and `splat` instructions into one that loads a single lane and duplicate it to all lanes of the vector.

(3) Load/store to variable pass is to eliminate repeated stack popping and pushing. Wasm `local.tee` instruction duplicates the top of the stack to a variable for later use. This pass applies this instruction to replace repeated load/store of the same memory address, and avoid the repeated stack pushing and popping of this variable. This can reduce kernel latency by ~7%.



(a) WASM **(b) WebGPU**

Figure 9: The MatMul kernel ([M,K,N]=[640,768,2304]) latency comparison of different primitive settings. The advanced setting is consistent across devices.

Just applying these light-weight optimization passes, the Wasm byte codes generated by NNJIT has no noticeable performance or byte code difference. The compiling latency can be accelerated by up to 125× compared to SOTA practice.

4.4 Compiling for WebGPU

In contrast to Wasm, which is executed in the browser's virtual machine, WebGPU implementation in browser essentially only translates WebGPU APIs into native GPU APIs (with limited optimization passes), while the target-related compilation is handled by the GPU driver. As illustrated in Fig. 6, it is not possible to eliminate the separated invocation of this target-related compilation; instead, we can only lower tensor IR to SPIR-V[26], a portable IR supported by both native GPU APIs and WebGPU. The reduction in compilation time for WebGPU is achieved through the kernel optimization space design, which will be discussed in Sec. 5.

5 ACCELERATE KERNEL TUNING WITH WEB-SPECIFIC LITE SPACE

To reduce the vast kernel optimization space, we propose the web-specific lite kernel space design based on two guidelines: the web specific requirements (Sec. 5.1), and the efficient utilization of hardware resources (Sec. 5.2). Existing worksalso aim to shrink kernel optimization space for inference on native hardware, but these spaces are either still too large to be evaluated online or require pre-defined hardware performance models. We will show that for in-browser inference, considering the two guidelines leads to a lite space size of just dozens, which can be evaluated online. Moreover, numerous edge devices offer the unique opportunity for crowd-sourcing global optimal kernels (Sec. 5.3).

5.1 Web-guided Offline Space Reduction

Web programming is aimed at achieving portability and security. For instance, both Wasm and WebGPU implement rigorous validation processes to prevent malicious or erroneous code, such as type errors, memory overflow, out-of-bounds access, and invalid jumps. These specialties convey consistent kernel performance patterns across devices. The related kernel implementations do not need to be evaluated on every device, which can significantly reduce the number of candidates within the kernel optimization space.

Performance pattern of Web programming. To illustrate the performance impact, Fig. 9 compares a MatMul latency with different primitive settings, *i.e.,* `cache_read` on and off for Wasm, and

the `unroll` on and off for WebGPU as examples. The performance shows the same pattern across devices. Disabled `cache_read` and enabled `unroll` always achieve better performance. What is more, they are also against the common setting for native kernels. The reasons are explained as follows.

The `cache_read` primitive creates a small buffer that can reside in different memory levels. As a nested loop in a kernel is mapped to various levels of tiling on the hardware. The small buffer can load a tile to improve data locality. For native kernel execution, the `cache_read` does improve performance on many devices. However, when it comes to Wasm kernels, the performance is reduced on all tested devices as shown in Fig. 9. The performance decrease is attributed to the costly Wasm validation process for memory allocation.

The `unroll` primitive explicitly unrolls the loop to reduce the loop related overheads. In native inference, the `unroll` primitive does not impact kernel performance on many devices, as the native GPU compiler can conduct loop unrolling optimization as needed. However, WebGPU only triggers a weak level of compiling optimization in native GPU to facilitate the quick response of web applications. As a result, the `unroll` primitive needs to be specifically set for tensor compiling to achieve better performance.

Discovery of Web-consistent primitive settings. Although we have demonstrated two typical examples of primitive settings, it remains challenging to discover all such primitives with cross-device consistent settings. To minimize human efforts, we propose developing a microbenchmark to automatically detect these primitives. The benchmarking is a one-time effort, as it is only related to the Web techniques used for backends, such as Wasm and WebGPU.

The microbenchmark suite automatically traverses all the primitives for a common-sized MatMul kernel (specifically with a shape of 4K×4K×4K in practice). The *one variable at a time* method is used to change the setting of only one primitive at a time, such as cache_read on/off. The suite is evaluated offline on multiple testing edge devices. We then compare the measurements across these testing devices. If the results are consistent, we set the primitives accordingly, *e.g.,* `cache_read` off and `unroll` on. Consequently, we can fix these settings when constructing the kernel optimization space, hence reducing the space. If the results are inconsistent, we consider them as device-dependent primitives. These will be processed in the device-guided online space construction module, allowing for adjustments based on specific device characteristics to optimize performance.

5.2 Device-guided Lite Space Building

Microbenchmark results remove the device-consistent settings from the kernel optimization space. The left ones are inconsistent across devices. This space is still large in the size of tens of thousands. This section will use formulated kernel hardware usage and heuristics to build the lite space with promising candidates for JIT evaluation on target devices.

Rational for heuristics and formulation. Tensors are divided to tiles for parallel execution. As shown in Fig. 10, the innermost loop tile is loaded to the registers by a thread. The second level tile is loaded to the L1 cache/shared memory by a block. For GPU programming, a block is a group of threads that are executed together on a core. The tile size prominently impacts the hardware utilization of a kernel implementation, and thus the kernel performance.

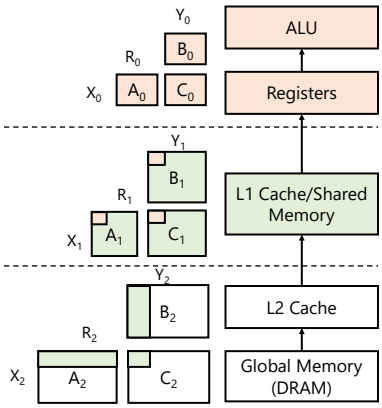


Figure 10: Tiles on the memory hierarchy for a MatMul.

Fundamentally, a tile size with efficient hardware utilization is to balance (1) the use of parallel computation units for fast computation and (2) the advanced memory storage *e.g.,* registers for fast data accesses. However, the two are normally conflicted with each other. More blocks and threads on a core can better saturate the parallel computing units and hide memory access stalls. However, since threads on a core share the registers and cache, too many threads may overuse the resources. On the other hand, fewer threads within the register and cache limit would under-use the parallel units. The sweet spot balancing the two highly depends on exact hardware, including the size of advanced memories, the computation and memory bandwidth, and the quality of compiling. It has to be evaluated on device to obtain.

Heuristics for efficient hardware utilization. We therefore formulate the hardware utilization based on the tile sizes, as shown in Table 1. These formulas require only the identification of the edge device type to ascertain hardware constraints, such as cache and register size. This device type could be obtained by Web programming interface. No other prior knowledge about devices are needed. By calculating the hardware utilization of each candidate and filtering the candidates by the heuristics, NNJIT can build the lite kernel optimization space for the hardware.

Taking the lite optimization space construction for WebGPU as an example, the tile size determines the utilization of registers, L1 cache, threads, and warps of a block. Given the total available registers, L1 cache, and warps of a GPU core, we can calculate the number of blocks that can share a GPU core ($Block_{core}$). To fully utilize the registers, Heuristic 7 guides the lite space to only include candidates whose number of blocks constrained by total registers to avoid wasting this fastest storage. Besides the hardware constraints heuristics (2, 3, 4 and 7), the Wasm and WebGPU specifications based heuristics (1, 5 and 6) are also considered.

The candidates that satisfy the heuristics will be in the optimization space. For online evaluation on devices, the candidates in the space will be evaluated in the order from the ones with max blocks per core to the ones with min blocks per core. Note in the actual calculation, there is a relaxation ratio for the hardware resources,

Table 1: Formulation of tile-based kernel hardware utilization and heuristics of web-specific lite space.

Params.: (Symbols follow Fig. 10) x_0, r_0, y_0: size of the inner-most tile loaded by a thread, x_1, r_1, y_1: size of the second tile loaded by a block, Reg_{thread}: number of registers used by a thread, $L1_{block}$: size of L1Cache or shared mem. used by a block, $Thread_{block}$: number of threads used by a block, $Warp_{block}$: number of warps used by a block. $Block_{core}$: number of blocks sharing a core	
Hardware resource usage: $Reg_{thread} = (x_0 \cdot r_0) + (r_0 \cdot y_0) + (x_0 \cdot y_0)$, $L1_{block} = (x_1 \cdot r_1) + (r_1 \cdot y_1) + (x_1 \cdot y_1)$, **For WebGPU**: $Thread_{block} = (x_1/x_0) \cdot (y_1/y_0)$, $Warp_{block} = Thread_{block}$/WARP_WIDTH, $Block_{core} = min$(RegLimit, L1Limit, WarpLimit), $RegLimit = \frac{TOTAL\_REGS\_PER\_CORE}{Reg_{thread} \cdot Thread_{block}}$, $L1Limit = \frac{L1\_SIZE\_PER\_CORE}{L1_{block}}$, $WarpLimit = \frac{TOTAL\_WARPS\_PER\_CORE}{Warp_{block}}$ **For WASM**: $Thread_{block} = 1, Block_{core} = 1$	
Heuristics for all	1. $SIMD\_width \leq 128$ 2. $x_0, r_0, y_0, x_1, r_1, y_1$: power of 2
Heuristics for WASM	3. $L1_{block} \leq L1\_SIZE\_PER\_CORE$ 4. Reg_{thread} = TOTAL_REGS_PER_CORE
Heuristics for WebGPU	5. $Thread_{block} \leq 256$ 6. $L1_{block} \leq 16\,KB$ 7. $Block_{core} = RegLimit$

since other variables in a kernel implementation also use registers or caches.

Finally, by applying both the Web-guided space reduction and the device-guided space building, our web-specific lite kernel optimization space only includes a few dozens of candidates, six orders of magnitude smaller than the naive search space, which is able to be evaluated online.

Potential kernel optimization for memory. Our current space design prioritizes latency considerations. We could also incorporate memory access or memory usage constraints into the heuristics for web-specific lite space. For instance, the search object could be the kernel implementation with the fewest memory accesses. Alternatively, we could constrain the value of $L1_{block}$ in Table 1, to reduce the usage of shared memory for WebGPU. Exploring extensions of nnJIT to other metrics represents a potential avenue for future work.

5.3 Crowdsourcing and Kernel Zoo

We have constructed a lite kernel optimization space. During deployment, we recognize a potential issue arising from the diverse nature of deployment environments, including various background workloads and hardware utilization levels. This may cause variance in assessed latency, potentially affecting our choice of the optimal kernel. To mitigate these concerns, we propose an *extended* kernel space.

Extended kernel space. For each device, we enhance the lightweight kernel space using an exploration-and-exploitation approach. The extended kernel space typically comprises two sets of candidates: 1) the exploration set, which includes the original lightweight kernel set and may be empty if optimal candidates for the device have already been discovered; 2) the exploitation set, obtained from the *crowdsourcing* module, which gathers and sends optimal kernel candidates to new devices with similar hardware specifications for further validation. Overall, the number of extended candidates is approximately one-tenth of the lite kernel space.

Table 2: The web-specific lite space for a MatMul kernel (M=K=N=4096) on WebGPU.

Primitives	Configures (symbols follow Table 1)
Cache Read	[Yes,No]
Cache Write	Yes
Reorder	$r_2, y_2, x_2, r_1, y_1, x_1, r_0, y_0, x_0$
Bind	$y_2 \rightarrow$ block.y, $x_2 \rightarrow$ block.x $y_1 \rightarrow$ thread.y, $x_1 \rightarrow$ thread.x
Unroll	y_0
Vectorize	x_0
Tile Size	$(r_0, y_0, x_0) \in [4, ..., 32]$; $(r_1, y_1, x_1) \in [32, ..., 256]$

Crowdsourcing and the kernel dataset. The diverse nature of web clients provides us with the opportunity to engage in *crowdsourcing*. The fundamental concept is that the searched optimal kernel implementations can be shared among devices with identical hardware. To facilitate this, we employ two designs: (1) we leverage the hardware ids as well as profiled hardware primitives as a criterion to ascertain whether devices can share the same generated optimal kernel. In particular, we form the primitive vector as $\vec{\rho} = \langle \rho_i \rangle, \rho_i \in \{0, 1\}$, where ρ_i denotes the i th primitive obtained from the micro benchmark. (2) In order to identify the best generated kernel, we adopt a majority voting strategy. Clients submit top-N (5 in our implementation) fastest implementations for a given kernel with the ranked weights. We also introduce the kernel dataset. We take as the primary index key $(t, s, \vec{\rho}, id)$, where t is the kernel type, s denotes the kernel shape, $\vec{\rho}$ is the primitive vector and id is the device id. Although the background noise is inevitable for latency measurements on edge devices, the majority voting result could reflect the likely kernel latency under the typical background workload.

6 IMPLEMENTATION

The Tensor-Web co-designed compilation pipeline in nnJIT is built upon TVM [12]. Particularly, for Wasm, we introduce a new compilation target (*i.e.,* Wasm IR) in TVM, by leveraging Binaryen [11] Wasm IR constructor. We develop the `WasmModuleNode` to enable lowering tensor intermediate representation (IR) to Wasm IR. We implement two crucial functions, `wasm::Builder` and `wasm::ModuleWriter`, to construct Wasm IR and compile Wasm binary. For WebGPU, we use the native GPU driver to compile.

To create the lite kernel space for nnJIT, we extend TVM by incorporating web-specific scheduling templates. In these templates, we set the configurations for web-consistent primitives and define the search space for device-dependent primitive configurations selected by heuristics. Table 2 shows an example lite kernel optimization space we build for a MatMul. We implement the microbenchmark via the evaluation MatMul kernel with the shape of 4 K×4 K×4 K, which are then compiled by the tool chain described above. We also adapt the in-browser inference runtime based from TVM, which is compatible to models in TF and ORT format.

To deploy nnJIT, we piggyback existing in-browser DNN deployment flow. Specifically, we introduce the kernel generation pipeline to the existing kernel distribution pipeline on the cloud side and incorporate kernel evaluation and kernel replacement function during inference on the edge side, Moreover, the cross-platform nature

Table 3: Evaluated kernels type and shape.

ID	Kernel Type	Kernel Size	Model
K0	MatMul	M=384,K=768,N=768	RoBERT
K1	MatMul	M=640,K=768,N=3072	GPT-2
K2	BatchMatMul	B=12,M=384,K=384,N=64	BART
K3	BatchMatMul	B=120,M=64,K=64,N=64	GPT-2

of web applications makes this integration a one-time effort for all devices. Overall, nnJIT comprises 2085 new lines of Python code, 1671 new lines of C++ code, and 564 new lines of JavaScript.

7 EVALUATION

7.1 Experiment Setup

Hardware and browsers. We conduct experiments on smartphones, laptops and desktops, a total of six devices, including Pixel 4, Vivo X30, Mate 20, Honor 70, SurfaceBook 3, Lenovo V9000, Honor MagicBook and HP EliteDesk equipped with ARM Cortex-A78/A77/A76/A73 CPU, AMD Ryzen 5800H CPU, Intel I9-12900 CPU, and NVIDIA RTX 3000/3070 Ti GPU, AMD Radeon GPU, Intel HD 630 GPU. We fix the maximum frequency on them to ensure consistent performance measurements. We evaluate nnJIT on 4 popular browsers: Chrome, Microsoft Edge, Firefox and Opera. All of them support Wasm, but only Chrome has solid WebGPU support. Without any specific indication, Chrome is the default browser used in the evaluation.

Kernels and models. We evaluate nnJIT on modern transformer models, including RoBERTa [30], BART [27], GPT-2 [34], T5 [35], and Llama 2 7B [41]. For the sequence-to-sequence models, such as GPT-2 and T5, we fix the input length at 384 to obtain the comparable results. We also evaluate the performance on typical kernels, picked from the models above, including MatMul and BatchMatMul with different shapes as listed in Table 3.

Baselines. We compare nnJIT with three in-browser DL inference frameworks as baselines, including TF.js (version 3.21.0), ORT-web (version 1.14.0) and pre-tuned AutoTVM [12]. For pre-tuned AutoTVM, we use the default kernel space, search algorithm *i.e.,* XGBoost and tuning trails *i.e.,* 1000 to generate and tune the kernels ahead-of-time. The pre-tuned target are Intel I7-10700 for Wasm and NVIDIA 3050 for WebGPU. They are excluded in our test devices.

Metrics. We use `performance.now()` function, a JaveScript API function to measure the latency of kernels and models running with Wasm, and `writeTimestamp` function of WebGPU API to measure the latency on WebGPU. Each kernel and model are evaluated with one warmup and 50 rounds, the averaged latency is reported. A round means one iteration of kernel generation (Fig. 4), starting with a selected candidate configuration and ending with the kernel generation and evaluation.

To measure memory consumption, we use `performance.memory.usedJSHeapSize` API function to catch the peak memory usage.

7.2 Overall Performance

Kernel performance cross devices. Fig. 11 demonstrates the latency of tested kernels on CPUs and GPUs, comparing baselines with nnJIT. On CPUs with Wasm, nnJIT achieves an average speedup of 4.59×. On GPUs with WebGPU, it accelerates kernel executions by an average of 2.77×. Specifically, nnJIT outperforms TF.js by 5.78× on Wasm and 1.68× on WebGPU. When compared to pre-tuned AutoTVM, the speedup is 6.43× on CPUs and 3.86× on GPUs. The inference speedup of nnJIT is mainly due to efficient kernel tuning for specific hardware, whereas *one-for-all* kernel approaches including TF.js, ORT-Web as well as pre-tuned AutoTVM, fall short in this regard.

Taking the WebGPU result as an example, pre-tuned AutoTVM behaves much worse on 3070Ti. This is because the pre-tuned AutoTVM kernels are tuned on NVIDIA 3050 (2560 cores), which obviously mismatches with 3070Ti (6144 cores). Therefore, K0 kernel of pre-tuned AutoTVM only activates 50% warps and utilize only 20% of L1 cache on 3070Ti.

Kernel performance cross browsers. Fig. 12 demonstrates the kernel speedup on four browsers. The actual speedups do have variance, but overall, the speedup pattern for each kernel is similar among browsers. nnJIT achieves an average speedup of 5.82× compared to baselines.

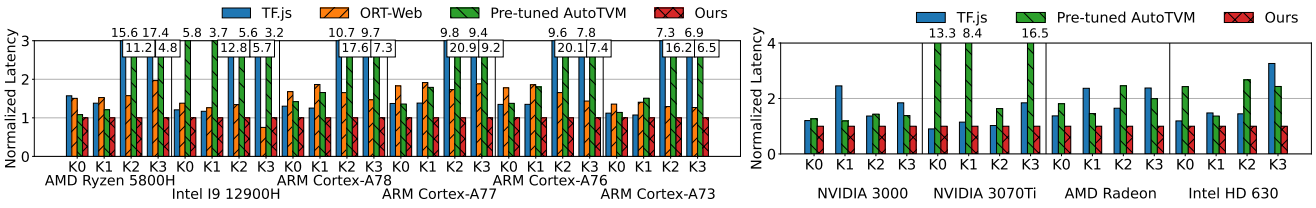
Kernel performance over tuning rounds. Fig. 13 showcases the kernel performance in GFLOPs over JIT tuning rounds on selected CPUs and GPUs. We use the K1 kernel configuration (Table3). nnJIT attains optimal performance on CPUs with Wasm after 10∼32 tuning rounds, while 25∼40 rounds are needed on GPUs. This can be attributed to the different sizes of web-specific lite spaces. Moreover, our compilation pipeline ensures that each tuning round takes only about 500ms for Wasm and 100ms for WebGPU.

We also compare nnJIT with the SOTA *one-for-each* kernel approach. We use AutoTVM and nnJIT to tune a kernel for the same hardware, shown in Fig. 14. The kernel configuration employed is K0 (Table3). On the Nvidia 3050, nnJIT reaches near-peak performance (1159 GFLOPs) within 25 rounds, while AutoTVM lags behind at 350 GFLOPs. On the Intel I7, nnJIT finds the best kernel implementation at the 8th round, demonstrating 2.80× speedup compared to AutoTVM at the same round. AutoTVM needs 1106 additional tuning rounds to achieve its optimal performance.

Model performance. We evaluate the end-to-end model performance achieved by nnJIT and other baselines. In Fig. 15, we denote RoBERTa as M0, BART as M1, GPT-2 as M2, and T5 as M3. As illustrated, nnJIT attains 2.76× and 1.37× speedup on average across the tested models compared to TF.js and ORT-Web on CPU with Wasm, respectively. Notably, on the AMD 5800H CPU, nnJIT improves by 8.27× on M3 compared to TF.js, while 5.64× on Intel I9. M2 (GPT-2) results on two mobile phones are missing because the model size is over the browser memory limitation of Android. As TF.js and ORT-Web cannot support all kernels in the tested models with WebGPU, we do not report their model latency.

Recently, Large Language Models (LLMs) have been gaining prominence. Thus, we evaluate Llama 2 7B model with nnJIT under 4-bit quantization. According to our evaluation, on Nvidia RTX 3000 using WebGPU, we achieve the processing speed of 12.16 tokens/s. Compared to the SOTA in-browser LLM, WebLLM [31], which operates at 8.72 tokens/s, nnJIT accelerates LLM in-browser inference by 39.4%.

We also show the model latency reduction over time, due to the JIT kernel optimization. As shown in Fig. 16, for BART, nnJIT takes 5.5 seconds to achieve the optimized latency for Nvidia 3000. For CPUs, peak performance is achieved after 17.8s and 22.1s for Intel I9 and ARM A76, respectively.



(a) WASM (b) WebGPU

Figure 11: Kernel latency executed with TF.js, ORT-web, pre-tuned AutoTVM as well as NNJIT on Chrome.

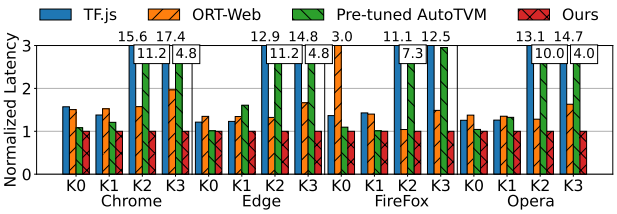
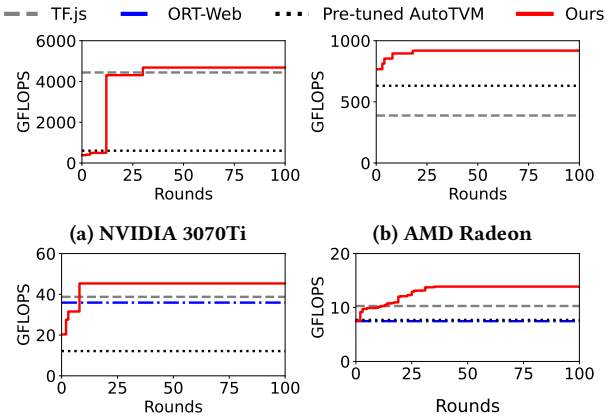


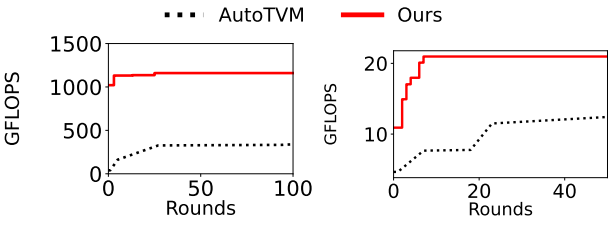
Figure 12: Kernel latency on four browsers with AMD Ryzen 5800H for WASM with NNJIT as well as baselines.



(a) NVIDIA 3070Ti (b) AMD Radeon

(c) Intel I9 12900H (d) ARM Cortex-A76

Figure 13: Kernel performance improvements with the JIT kernel optimization rounds on different devices.



(a) NVIDIA 3050 (b) Intel I7 10700

Figure 14: Kernel performance improvements along with the JIT tuning rounds with NNJIT and AutoTVM.

Memory. We evaluate the peak memory consumption in Table 4. The peak memory of NNJIT is, on average, 55.7% and 49.6% less compared to TF.js and ORT-Web, respectively. Compared with pre-tuned AutoTVM, NNJIT only increases 2.2% peak memory for JIT kernel optimization.

Table 4: Peak memory consumption (MB) executed with TF.js, ORT-web, pre-tuned AutoTVM as well as NNJIT on Chrome.

Models	TF.js	ORT-Web	Pre-tuned AutoTVM	Ours
M0	1484	1196	536	550
M1	N/A	1354	642	656
M2	2293	1842	636	652
M3	398	368	263	268

Table 5: The compiling latency and achieved kernel latency of NNJIT with different optimization passes.

	Compilation Latency (sec)	Kernel Latency (ms)
Conventional Compilation	5.8~62.9	76
Ours w/o opt. passes	0.4~0.5	234
Ours w/ offset load/store	0.5~0.6	88
Ours w/ offset load/store & combined instruction	0.5~0.6	82
Ours w/ offset load/store & combined instruction & load/store to variable	0.5~0.6	74

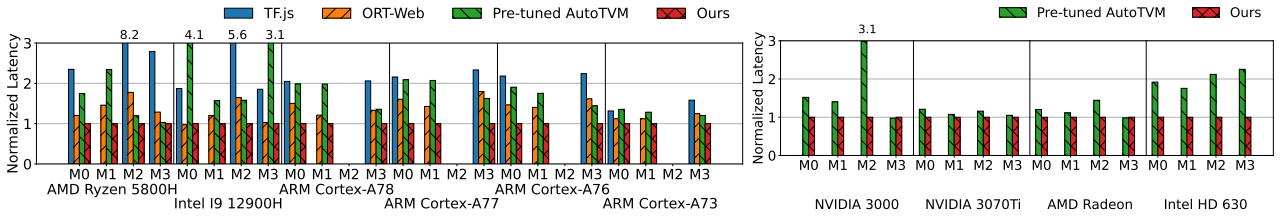
Table 6: Kernel space of AutoTVM and NNJIT.

Kernel Type (Size)	AutoTVM		NNJIT	
	WASM	WebGPU	WASM	WebGPU
MatMul (M=384,K=768,N=768)	2,099,520	42,768,000	10~32	41
BatchMatMul (B=12,M=384,K=384,N=64)	2,694,384	74,131,200	10~32	30

7.3 Ablation Study

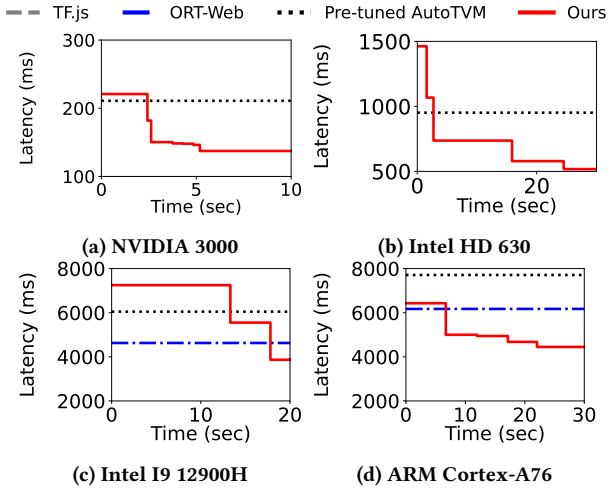
Tensor-Web co-designed compilation. Table 5 presents the compiling latency and achieved kernel latency for both the baseline and NNJIT on AMD 5800H CPU. We use AutoTVM's conventional compilation pipeline as the baseline. For our optimized pipeline, we examine three optimization passes, namely offset load/store, combined instruction, and load/store to variable pass, to assess their individual contributions to the kernel latency reduction. The same kernel implementation is used for all cases.

As demonstrated, our compilation pipeline with all optimizations is over up to 125.8× faster than the baseline, while maintaining a similar kernel inference latency (76ms and 74ms). Furthermore, our pipeline with three optimization passes results in a significant performance improvement of 166%, 185%, and 216%, respectively, with the compiling overhead increasing by 25%.



(a) WASM (b) WebGPU

Figure 15: Model latency executed with TF.js, ORT-web, pre-tuned AutoTVM as well as NNJIT.



(a) NVIDIA 3000 (b) Intel HD 630

(c) Intel I9 12900H (d) ARM Cortex-A76

Figure 16: Model performance improvements with the JIT kernel optimization time on different devices.

Web-Specific lite space. Table 6 compares the size of our lite kernel optimization space and AutoTVM for two typical kernels, MatMul and BatchMatMul.Notably, the web-specific lite kernel space size is, on average, around 0.0013% and 0.000068% of the AutoTVM space on Wasm and WebGPU, respectively, decreasing search candidates from millions to dozens. In combination with our optimized compilation pipeline, NNJIT reduces the overall kernel generation cost from hours to seconds, enabling JIT-powered inference in web browsers.

Overhead. NNJIT enables JIT kernel optimization with minimal overhead. For example, the microbenchmark is a one-time effort executed in the offline stage, taking less than 1 second on a AMD Ryzen 5800H CPU according to our measurements. During JIT inference, kernels are sequentially pushed from the server to devices. The compiled kernel sizes range between 5~30KB, which does not add a significant burden to the network load. To evaluate the newly arriving kernels, a device typically takes 69~728ms for most kernels based on our assessment, which is nearly imperceptible.

8 RELATED WORKS

DL kernel generation. Many works [7–9, 12, 29, 40, 46] focus on automatically searching and generating optimal kernel implementations from a vast space. TVM [12] generate DL kernels based on the space of manual schedule templates and a learned cost model to search for the best kernel implementation. Ansor [46] and Antares [7] generates higher-performance DL kernels than TVM without manual schedule templates and reduces the average search time. Romou [29] supports new primitives to generate mobile-GPU-friendly DL kernels and accelerates kernel generation through hardware-aware search space pruning. Although this approach can reduce the space by 99%, the number of remaining candidates is still on the order of 10K. Triton [40] is a DL kernel generator that extends LLVM IR and adds an additional tile-level optimization pass, achieving high DL kernel performance. OpenXLA [9] is a ML compiler ecosystem that aims to simplify and accelerate ML development by addressing fragmentation between different ML frameworks and hardware. IREE [8] is an end-to-end compiler and runtime toolkit specifically for machine learning (ML) models. Compared to NNJIT, these works are not for in-browser inference. Besides, the last three works do not really support auto kernel optimization. For example, Triton JIT compiler can only tune CUDA kernels within a space that is designed by experts.

In-Browser DL inference. The emergence of DL frameworks, such as TensorFlow.js [39] and ONNX Runtime Web [45], has significantly contributed to making in-browser DL inference a reality. TensorFlow.js from Google supports JavaScript, Wasm, WebGL, and WebGPU. ONNX Runtime Web from Microsoft, facilitates in-browser DL inference by processing models in ONNX format. However, it only supports Wasm and WebGL backends. Transformer.js [24] is recently released to support the transformer model inference in browsers. These frameworks all uses the pre-defined kernels, leading to suboptimal performance across edge devices as discussed in Sec. 2.2. NNJIT stands out as the first in-browser inference framework that enable JIT compilation, thereby ensuring peak performance across devices. Furthermore, NNJIT automatically generates kernels, which allows for the support of the new kernels and models with minimal manual effort.

9 CONCLUSION

In this paper, we present NNJIT, the first in-browser inference system that enables JIT optimized kernel generation, supporting Wasm and WebGPU. Our evaluation shows that NNJIT accelerates inference by an average of 4.44× compared to TF.js, ORT-Web, and AutoTVM, while maintaining minimal compilation overhead.

REFERENCES

[1] Retrived in June, 2023. Brain.js: GPU accelerated Neural networks in JavaScript for Browsers and Node.js. https://brain.js.org/.

[2] Retrived in June, 2023. The LLVM Compiler Infrastructure. https://llvm.org/.

[3] Retrived in June, 2023. WebAssembly. https://webassembly.org/.

[4] Retrived in June, 2023. WebDNN. https://mil-tokyo.github.io/webdnn/.

[5] Retrived in June, 2023. WebGL: 2D and 3D graphics for the web. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[6] Retrived in June, 2023. WebGPU-W3C Working Draft. https://www.w3.org/TR/webgpu/.

[7] Retrived in March, 2024. Antares. https://github.com/microsoft/antares.

[8] Retrived in March, 2024. IREE. https://iree.dev/.

[9] Retrived in March, 2024. OpenXLA. https://github.com/openxla.

[10] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rygG4AVFvH

[11] Binaryen. 2023. https://github.com/WebAssembly/binaryen

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[13] Brandon Jones Corentin Wallez and François Beaufort. 2023. WebGPU: Unlocking modern GPU access in the browser. https://developer.chrome.com/blog/webgpu-io2023/#:~:text=WebGPU%20is%20the%20successor%20to,GPU%20capabilities%20in%20the%20future.

[14] cuBLAS. 2023. https://docs.nvidia.com/cuda/cublas/index.html

[15] Eigen. 2023. https://eigen.tuxfamily.org/index.php

[16] Satya Jandhyala Emma Ning, Yulong Wang. 2024. ONNX Runtime Web unleashes generative AI in the browser using WebGPU. https://cloudblogs.microsoft.com/opensource/2024/02/29/onnx-runtime-web-unleashes-generative-ai-in-the-browser-using-webgpu/

[17] Emscripten. 2023. https://emscripten.org/docs/introducing_emscripten/about_emscripten.html

[18] V8 Engine. 2023. https://v8.dev/

[19] OpenGL ES. 2023. https://www.khronos.org/opengles/

[20] Tao Ge, Ting Cao, Siqing Chen, and Qiong Ning. 2023. Achieving Zero-COGS with Microsoft Editor Neural Grammar Checker. https://www.microsoft.com/en-us/research/blog/achieving-zero-cogs-with-microsoft-editor-neural-grammar-checker/.

[21] Google. 2024. Why On-Device Machine Learning? https://developers.google.com/learn/topics/on-device-ml/learn-more

[22] Thales Group. 2023. Cloud Assets the Biggest Targets for Cyberattacks, as Data Breaches Increase. https://www.thalesgroup.com/en/worldwide/security/press_release/cloud-assets-biggest-targets-cyberattacks-data-breaches-increase.

[23] WebAssembly Community Group. 2023. WebAssembly Specification Release 2.0.

[24] Huggingface. 2023. https://huggingface.co/docs/transformers.js/index

[25] JavaScript. 2023. https://www.w3.org/standards/

[26] Khronos. 2023. SPIR overview. https://www.khronos.org/spir/

[27] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *CoRR* abs/1910.13461 (2019). arXiv:1910.13461 http://arxiv.org/abs/1910.13461

[28] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2022. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. In *MobiSys '22: The 20th Annual International Conference on Mobile Systems, Applications and Services, Portland, Oregon, 27 June 2022 - 1 July 2022*. ACM, 261–272. https://doi.org/10.1145/3498361.3538922

[29] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. 2022. Romou: Rapidly Generate High-Performance Tensor Kernels for Mobile GPUs. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking* (Sydney, NSW, Australia) *(MobiCom '22)*. Association for Computing Machinery, New York, NY, USA, 487–500. https://doi.org/10.1145/3495243.3517020

[30] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[31] Web LLM. Jul, 2023. Web LLM. https://github.com/mlc-ai/web-llm.

[32] Yun Ma, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. 2019. Moving Deep Learning into Web Browser: How Far Can We Go?. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 1234–1244. https://doi.org/10.1145/3308558.3313639

[33] OpenGL. 2023. https://www.opengl.org/

[34] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[36] Statista. 2023. https://www.statista.com/topics/5684/web-browsers/#topicOverview

[37] Steam. 2023. https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam

[38] TensorFlow. 2023. https://www.tensorflow.org/

[39] TensorFlow.js. 2023. https://www.tensorflow.org/js

[40] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[42] Vulkan. 2023. https://www.vulkan.org/

[43] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *ACM MobiCom '21: The 27th Annual International Conference on Mobile Computing and Networking, New Orleans, Louisiana, USA, October 25-29, 2021*. ACM, 215–228. https://doi.org/10.1145/3447993.3448625

[44] Qipeng Wang, Shiqi Jiang, Zhenpeng Chen, Xu Cao, Yuanchun Li, Aoyu Li, Ying Zhang, Yun Ma, Ting Cao, and Xuanzhe Liu. 2024. Exploring the Impact of In-Browser Deep Learning Inference on Quality of User Experience and Performance. arXiv:2402.05981 [cs.LG]

[45] ONNX Runtime Web. 2023. https://onnxruntime.ai/docs/tutorials/web/

[46] 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. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[47] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508

[48] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu