Accelerating In-Browser Deep Learning Inference on Diverse Edge Clients through Just-in-Time Kernel Optimizations

Fucheng Jia[1,3,*], Shiqi Jiang[1], Ting Cao[1,†], Wei Cui[1], Tianrui Xia[1,*], Xu Cao[1], Yuanchun Li[2], Deyu Zhang[3], Ju Ren[2], Yunxin Liu[2], Lili Qiu[1], Mao Yang[1]

[1] Microsoft Research [2] Tsinghua University [3] Central South University

ABSTRACT

Web applications are increasingly becoming the primary platform for AI service delivery, making in-browser deep learning (DL) inference more prominent. However, current in-browser inference systems fail to effectively utilize advanced web programming techniques and customize kernels for various client devices, leading to suboptimal performance.

To address the issues, this paper presents the first in-browser inference system, nn-JIT.web, which enables just-in-time (JIT) auto-generation of optimized kernels for both CPUs and GPUs during inference. The system achieves this by using two novel web programming techniques that can significantly reduce kernel generation time, compared to other tensor compilers such as TVM, while maintaining or even improving performance. The first technique, *Tensor-Web Compiling Co-Design*, lowers compiling costs by unifying tensor and web compiling and eliminating redundant and ineffective compiling passes. The second technique, *Web-Specific Lite Kernel Optimization Space Design*, reduces kernel tuning costs by focusing on web programming requirements and efficient hardware resource utilization, limiting the optimization space to only dozens.

nn-JIT.web is evaluated for modern transformer models on a range of client devices, including the mainstream CPUs and GPUs from ARM, Intel, AMD and Nvidia. Results show that nn-JIT.web can achieve up to 8.2× faster within 30 seconds compared to the baselines across various models.

1 INTRODUCTION

Web applications are increasingly becoming the primary means to deliver AI services, such as ChatGPT [2], StableDiffusion [4], Web LLM [23] and the suite of AI services within M365 for Web [3]. This AI deployment shift is attributed to the compelling advantages of Web applications, including: cross-platform execution, that a Web application can run on any device with a browser ; *click and run* deployment, with no need for installation; and simplicity of maintenance, that application updates can be timely available to users.

† Ting Cao is the corresponding author.
* Fucheng Jia and Tianrui Xia were research interns at Microsoft Research.

With such shift, there is the surge in interest towards performing DNN inference directly within Web browsers, *i.e.,* in-browser inference. In-browser inference can provide a more responsive user experience and enhanced privacy protection by avoiding round-trips to the cloud, as well as reduce the expense of cloud computing resources for serving a large number of clients. In-browser inference is made viable by the continuous advances in Web programming techniques, such as the recently introduced WebAssembly (abbreviated Wasm) [5] and WebGPU [8], as well as the fast-growing computing capabilities of client devices.

However, current in-browser inference systems, such as TensorFlow.js[29], ONNX Runtime Web[32], WebDNN[6], and brain.js[1], suffer from two major drawbacks, leading to inferior performance. Firstly, these systems lag behind advanced web programming techniques, as they require handwritten kernels for each web programming backend *e.g.,* JavaScript, Wasm, WebGL[7]. Integrating a new backend necessitates significant rewriting efforts, resulting in limited support for emerging technologies like WebGPU[11]. Secondly, their predefined kernels do not account for hardware diversity, causing a *one-for-all* approach that delivers poor performance across various client devices. As we will show in the paper, our proposed device-customized kernels demonstrate a potential speed-up of several times.

To address these challenges, tensor compiling techniques such as TVM[10], Ansor[34] and FlexTensor [35] can be employed to automatically generate customized kernels without manual efforts. However, tensor compilers necessitate *ahead-of-time* kernel generation for known hardware, due to the hours even days of kernel generation cost and the requirement of on-device kernel evaluation. This approach is more practical for limited target devices, such as those in cloud environments. Unfortunately, Web applications are intentionally designed to operate on a wide range of hardware and software environments, encompassing diverse CPUs, GPUs, and OS. Generating kernels ahead-of-time for each hardware is impractical. Therefore, achieving optimal in-browser inference performance for each client device without manual intervention remains an unresolved challenge.

To tackle it, we rethink the specialties of Web. Compared to native 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 device.

Based on this insight, we present nn-JIT.web, the first in-browser DNN inference system with the unique ability to automatically generate and continuously improve customized kernels during inference for target 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 key challenge lies in enabling JIT generation of optimized kernels, a feat that has never been accomplished before. Current tensor compilers perform *compiling* and *kernel tuning* processes to identify reasonable kernels. Tensor computations are implemented as nested multi-level loops to compute each tensor element. Various loop arrangements, such as different tiling sizes, unrolling factors, and loop orders, result in a kernel optimization space. The kernel tuning process iteratively selects and evaluates potential candidates from this space to find optimal ones. The evaluation of each candidate invokes the compiling process to generate executable codes and run on the target device.

As discussed in related papers [21, 36], the lengthy time required to generate optimized kernels is due to 1) the compiling cost and 2) the vast kernel optimization space. Compiling each candidate can take minutes, as numerous transforming passes are needed for both tensor level and target language level, *e.g.,* Wasm. The extensive optimization space prolongs the kernel tuning process in searching for reasonable candidates. To reduce the space, Romou[21] eliminates candidates that overuse hardware resources. Although this approach can reduce the space by 99%, the number of remaining candidates is still on the order of 10K. Roller[36] selects promising candidates by building a hardware performance model for known hardware, which is impractical for the diverse client devices found in Web environments.

nn-JIT.web can facilitate JIT generation of optimized kernels based on our key findings of Web programming, that can reduce the compiling cost and kernel optimization space. 1) Web programming interface is designed with simple instruction sets and execution model for running 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 target-language (*i.e.,* Wasm) level compiling, nn-JIT.web employs a unified compiling pipeline from tensor IR (Intermediate Representation) directly to Wasm IR, which completely eliminates the required invocation of LLVM Wasm backend or Emscripten [14] for separated Wasm compiling. The unified pipeline co-designs the tensor and Wasm compiling optimizations to avoid redundant and ineffective ones. The optimizations in LLVM Wasm backend is the best covered in the kernel optimization space. Only the optimizations closely related to Wasm instructions are kept to apply on the Wasm IR. This new compiling pipeline dramatically reduces the cost per candidate, from minutes to milliseconds.

The second technique is *Web-specific lite kernel optimization space design*, guided by two principles: Web programming requirements and efficient utilization of hardware resources. As Web requirements cause consistent performance patterns across devices, to identify their impact on the kernel optimization space, we compose a microbenchmark suite that traverses the tensor compiling primitives (code transformations conducted on tensor IR to generate kernels) such as loop order and unroll, in a *one-variable-at-a-time* manner. The suite is evaluated offline to identify the efficient primitive configurations. This can reduce the space size to tens of thousands. The hardware utilization is mostly decided by the tile sizes of a kernel implementation. An efficient hardware utilization requires the tile size to balance the contention between improved parallel hardware execution and reduced low-level memory accesses. This is *inconsistent* across hardware depending on the hardware specs. We therefore use the heuristic-based method to select promising tile sizes to be in the kernel optimization space, to evaluate on the target device during JIT. By the guidelines, the number of candidates in space is reduced to only dozens.

Based on the two techniques, we develop the nn-JIT.web. After the initial model and kernels are downloaded to run on the target client device, nn-JIT.web generates the lite kernel optimization space. Candidates in the space are compiled one-by-one using our unified compiling pipeline and evaluated on the client 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 is also crowdsourced from ones with similar hardware to achieve optimal kernels much faster.

nn-JIT.web is implemented for both Wasm for CPUs and WebGPU for GPUs. Wasm is already supported by mainstream browsers, and CPUs are ubiquitous in client devices; thus, we prioritize Wasm support. WebGPU, although still in its early stages, shows great promise. Thanks to our JIT kernel generation, nn-JIT.web is the first general inference system that supports WebGPU for complex models, serving as a strong showcase for our advantages.

nn-JIT.web is evaluated on representative transformer models, with suitable size to run in-browser on client devices, including encoder model RoBERTa [22], encoder-ecoder model BART [20] and T5 [27], and decoder model GPT-2 [26]. Evaluation platforms cover a range of mobile and desktop hardware, including ARM CPU (Cortex-A76 and 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 that within 30 seconds, nn-JIT.web can achieve average 26.65 times faster kernels compared to the baseline, and 2.36 times faster model inference.

To summarize, our main contributions include:

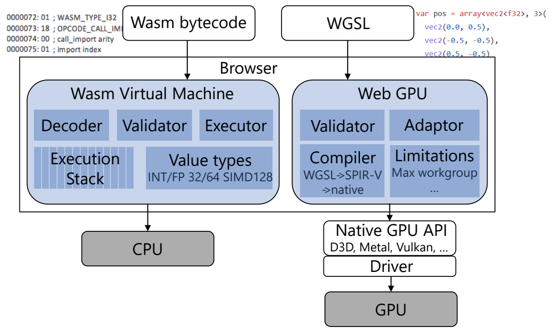
- This paper proposes the first in-browser inference system that enables JIT optimized kernel generation.
- The Tensor-Web compiling co-design avoids the ineffective and redundant optimizations, reducing the compiling cost from minutes to milliseconds.
- The Web-specific lite kernel space design is guided 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 client devices, achieving up to 8.2× speedup, compared to SOTA inference frameworks.

2 BACKGROUND AND MOTIVATION

2.1 DL Inference in Web Browsers

Enabling DL inference in modern Web browsers is nontrivial [24]. 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 [13] for the CPU and cuBLAS [12] for the GPU.

To make DL inference in browsers possible, alternative programming interfaces, hence backends, are proposed to use. JavaScript [18] is firstly leveraged to implement DL kernels and graphs in Web DL frameworks [29]. JavaScript has no-static data type and no vectorization support. Although



```
0000072: 01 ; WASM_TYPE_I32
0000073: 18 ; OPCODE_CALL_IM
0000074: 00 ; call_import arity
0000075: 01 ; import index
```

```
var pos = array<vec2<f32>, 3>(
  vec2(0.0, 0.5),
  vec2(-0.5, -0.5),
  vec2(0.5, -0.5)
```

Figure 1: The Wasm and WebGPU support in browser.

some efforts like V8 Engine [15] could significantly accelerate JavaScript code, the DL execution with it is still extremely inefficient in JavaScript environment.

To cope with it, WebAssembly (Wasm) [5] 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 could also be utilized within browsers. For instance, WebGL has been integrated in TensorFlow.js. WebGL provides a set of JavaScript interfaces to access GPU that originally enable rendering 3D graphics on Web pages. It is based on OpenGL ES 2.0 [16], a subset of OpenGL [25]. Thus, certain functionalities are not available. Meanwhile, as a rendering library, it failed to utilize the computation pipelines in modern GPUs due to limited instructions for computation.

To unleash the power of GPU, WebGPU, the successor of WebGL, is proposed. In addition to graphics rendering, 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 to run, such as Vulkan [31]. For portability, WebGPU also specifies limitations for the hardware usage. The validator is again to check the kernel for security.

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 DL

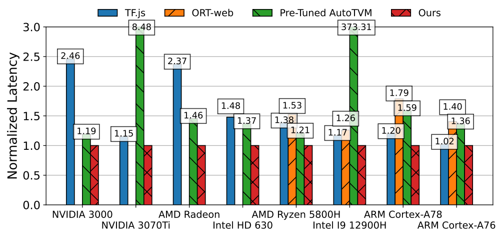


Figure 2: The normalized kernel latency of handwritten, pre-tuned, and our online-tuned MatMul kernels ([M,K,N]=[640,768,2304]) on eights devices.

kernels shipped within these frameworks are usually handwritten or ported from native DL frameworks, *e.g.,* TensorFlow [28]. To optimize the kernels, DL compilers such as TVM [10] are also extended to support generating and optimizing kernels implemented in Wasm and even WebGPU automatically. However, generating kernels for Web usually takes significantly long time, *e.g.,* nearly 2 hours for one Matrix Multiplication (MatMul) kernel. Besides that, the performance of tuned kernels are almost far from the optimal. We would discuss the issue in detail in the followings.

2.2 Inference Performance Issues

To understand in depth the DL inference performance in browsers, we conduct the preliminary study, specifically we measure the inference latency of a typical DL kernel, MatMul, to demonstrate the potential performance issues for DL inference in browsers. We have the following observations:

The one-for-all kernels are suboptimal across devices. Web applications are running on millions of 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 the way of 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 146% when compared to customized ones. Similarly, without tuning, the generated kernel from the tensor compiler TVM shows a slowdown of 19% to

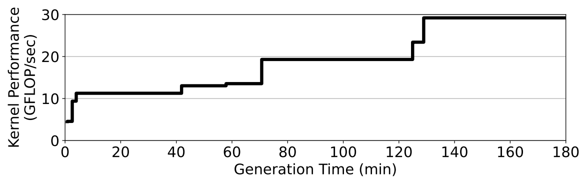


Figure 3: The generated MatMul kernel ([M,K,N]=[640,768,2304]) performance and generation time of TVM on AMD 5800H CPU.

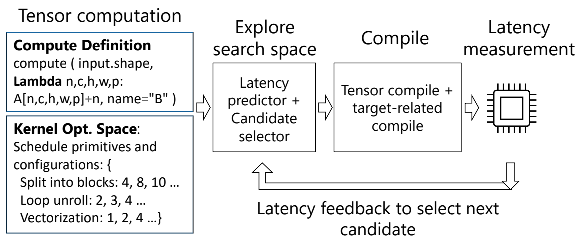

Tensor computation
Compute Definition
compute (input.shape,
Lambda n,c,h,w,p:
A[n,c,h,w,p]+n, name="B")
Kernel Opt. Space:
Schedule primitives and configurations: {
Split into blocks: 4, 8, 10 ...
Loop unroll: 2, 3, 4 ...
Vectorization: 1, 2, 4 ...}
Explore search space
Latency predictor + Candidate selector
Compile
Tensor compile + target-related compile
Latency measurement
Latency feedback to select next candidate

Figure 4: A common tensor compiler pipeline.

multiple times, depending on devices. These results highlight the need for customized kernels tailored to each 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 optimized 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 optimized kernel generation cost is due two primary causes: the exceedingly large kernel optimization space and the bloated tensor compilation process.

Fig. 4 illustrates a common tensor compiler pipeline. For a tensor compiler, 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 for the tensor IR *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 IR transformations to generate 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

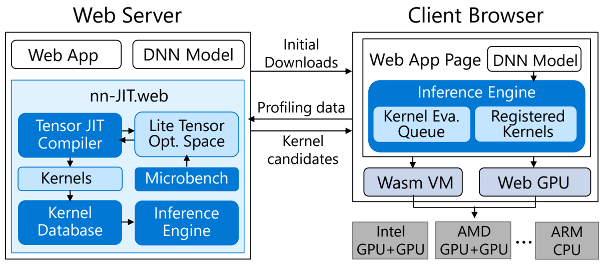


Figure 5: Overview of nn-JIT.web.

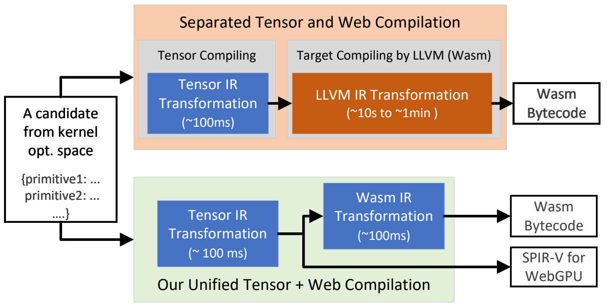

Separated Tensor and Web Compilation
Tensor Compiling
Target Compiling by LLVM (Wasm)
Tensor IR Transformation (~100ms)
LLVM IR Transformation (~10s to ~1min)
Wasm Bytecode
A candidate from kernel opt. space
{primitive1: ... primitive2: ... .}
Tensor IR Transformation (~ 100 ms)
Wasm IR Transformation (~100ms)
Wasm Bytecode
SPIR-V for WebGPU
Our Unified Tensor + Web Compilation

Figure 6: Our unified Tensor+Web compiling compared to conventional separated Tensor and Web compiling.

is around 42 M. Smart searching algorithms and hardware performance models are normally employed to only select promising candidates to conduct actual compilation and evaluation on the target device. Even so, thousands of candidates are generally needed to be 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 optimized kernel generation cost will be hours.

Therefore, to reduce the optimized kernel generation cost for JIT, we need to reduce the compiling cost for each candidate, and reduce the number of candidates in the space. To achieve this, we propose nn-JIT.web. In the following sections, we will introduce the design principles and key techniques of nn-JIT.web.

3 NN-JIT.WEB OVERVIEW

Fig. 5 is the overview of nn-JIT.web. It consists of four modules: the *tensor 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 customized kernels tailored to known devices; The whole kernel generation and inference process facilitated by nn-JIT.web 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 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 JIT compiler* of nn-JIT.web 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*. Furthermore, compared to cloud inference, which necessitates uploading raw user input data, only performance profiles are sent to the server. This approach aligns with common Web practices to enhance user experience and protect privacy.

The online kernel generation combined with JIT-styled inference ensures optimal performance on Web. 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 compiling (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 compiling. As shown in Fig. 6, the conventional compiling process of tensor compilers 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 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 compiling generates the loop, a separate target-related compiling library such as LLVM is normally invoked to generate the executables. As these target-related compiling libraries target to compile any general-purpose programs, there are many compiling passes, taking long time to complete.

Specifically, for Wasm as shown in Fig. 1, the target execution environment is the Wasm virtual machine running within the browser. The target-related compiling library is LLVM or Emscripten to compile tensor-level loop to Wasm bytecode. The Wasm-related compiling by LLVM/Emscripten also contributes the majority of total tensor compilation cost.

Feasibility of eliminating target-related compiling. We therefore explore the possibility to eliminate this target-related compiling, 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 [17], for the purpose of easy decoding, running efficiency, and security. Consequently, many sophisticate 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 compiling optimizations. For example, the unrolled loop generated by LLVM optimization pass is very likely also included in the kernel optimization space, which will be evaluated as well. The separated tensor and Wasm compiling cannot avoid the redundancy. In addition, the tensor computation defined by

```
@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.

the tensor domain specific languages does not need the complex compiling optimizations for 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 tensor compiling pipeline, which unifies the tensor and Wasm compiling as shown in Fig. 6. It removes the separated target compiling invocation, and compiles tensor IR directly to the target executables *e.g.,* 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 converting 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 Wasm IR. Wasm has only been lowered from LLVM IR before, which is also a lower level IR, facilitating the transformation. For example, LLVM has already lowered the high-level `for` statement.

Algorithm 1: Lower Tensor IR to Wasm IR for loop

```
   input  : ForNode of Tensor IR $forNode$
   output : LoopExpression of Wasm IR $loopExpr$
1      ▷ for(loopVar=begin;loopVar<end;loopVar+=stride) body;
2  $loopVar \leftarrow$ createWasmVar();
3  $initLoopVarExpr \leftarrow$ makeLocalSet($loopVar$, $forNode$.begin);
4  $ltExpr \leftarrow$ makeBinary(Op::Lt, $loopVar$, $forNode$.end);
5  $brIfExpr \leftarrow$ makeBreak($loopVar$.label, $ltExpr$);
6  $addLoopVarExpr \leftarrow$ makeBinary(Op::Add, $loopVar$, $forNode$.stride);
7  $bodyExpr \leftarrow$ VisitStmt($forNode$.body);
8  $innerExpr \leftarrow$ makeBlock($bodyExpr$, $AddLoopVarExpr$, $brIfExpr$);
9  $loopExpr \leftarrow$ makeLoop($loopVar$.label, $innerExpr$);
10 $loopExpr \leftarrow$ makeBlock($initLoopVarExpr$, $loopExpr$);
```

Fig. 7 uses a code snippet to illustrate the differences between the two IRs by using the MatMul implementation as an example. The tensor IR is represented as a sequence of statements, such as the `for` loop statement. Wasm, 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 is implemented as a stack-based machine, 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.

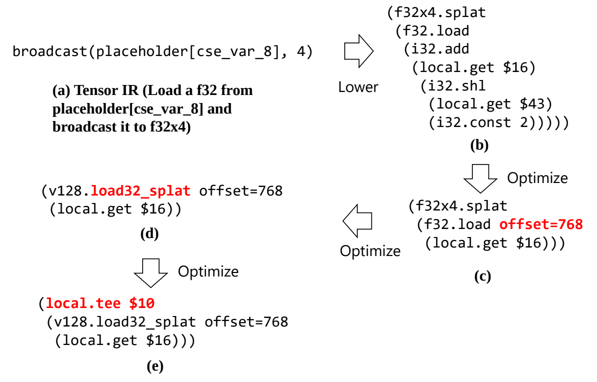
Map `for` statement to an expression block. To map the tensor IR to Wasm, we traverse the tensor IR AST (abstract Syntax Tree) to transform the sequence of statements to sequence of expressions. The difficulty lies in handling the `for` statement. Since Wasm does not utilize statements, we construct a nested sequence of expressions as a block enclosed by the Wasm `loop&end` instructions for this statement.

As we show in Algorithm 1, the sub-expressions, *e.g.*, loop variable calculation, are created while traversing the `for` node of the tensor IR AST (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.

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.


Lower
Optimize
Optimize
Optimize
(a) Tensor IR (Load a f32 from placeholder[cse_var_8] and broadcast it to f32x4)
(b)
(c)
(d)
(e)

```
broadcast(placeholder[cse_var_8], 4)
```

```
(f32x4.splat
 (f32.load
  (i32.add
   (local.get $16)
    (i32.shl
     (local.get $43)
     (i32.const 2)))))
```

```
(f32x4.splat
 (f32.load offset=768
  (local.get $16)))
```

```
(v128.load32_splat offset=768
 (local.get $16))
```

```
(local.tee $10
 (v128.load32_splat offset=768
  (local.get $16)))
```

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.

Wasm provides the offset augmented load/store instruction to avoid the address calculation. This pass is to utilize this offset. 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. The variable also resides in the linear memory. This pass applies this instruction to replace repeated load/store of the same memory address, to avoid the repeated stack pushing and popping. This can reduce kernel latency by ~7%.

Just applying these light-weight optimization passes, the Wasm byte codes generated by nn-JIT.web 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

For WebGPU, the target compilation platform is the hardware GPU. In contrast to Wasm byte code, 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-specific compilation is handled by the GPU driver. As illustrated in Fig. 6, it is not possible to eliminate the separate invocation of target-specific compilation; instead, we can only lower tensor IR to SPIR-V[19], a portable IR supported by both native GPU APIs and WebGPU. The reduction in compilation time for WebGPU is achieved through the tensor 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 works [21, 36] also aim to shrink kernel optimization space for inference on native hardware. However, 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 a few dozens, which can be evaluated online. Moreover, the numerous web application clients offer the unique opportunity for crowdsourcing the global optimal kernel (Sec. 5.3).

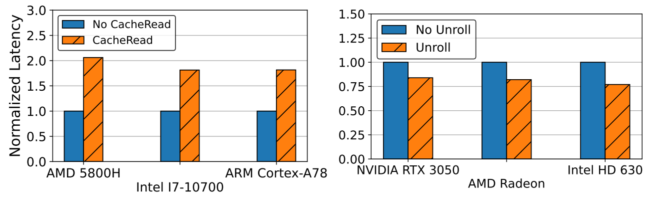
5.1 Web-guided Offline Space Reduction

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. This decrease in performance 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



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

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

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 client 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 search space, hence reducing the space. If the results are inconsistent, we consider them as device-dependent primitives. These would 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. A tensor computation is mapped to a kind of loop arrangement in a kernel implementation, and further mapped to tiles in

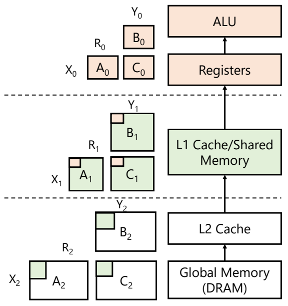


Figure 10: Tiles on the memory hierarchy for a MatMul.

the hardware during execution. As shown in Fig. 10, the innermost loop tile is mapped to the registers, and the second level loop tile is mapped to the L1 cache/shared memory. Therefore, the tile size prominently decides the hardware usage of the kernel implementation, and thus the kernel performance.

Fundamentally, a tile size with efficient hardware usage is to balance 1) the use of parallel computation units for fast computation and 2) the advanced memory storage *e.g.,* shared memory for fast data accesses. However, the two are normally conflicted with each other. More threads can better saturate the parallel computing units and hide memory access stalls. However, more threads can overuse the registers and shared memory. On the other hand, threads that satisfy the registers can under use the parallel units. The sweet spot to balance the two depends on the target hardware and running environment, such as the size of advanced memories, the computation and memory bandwidth, and the quality of compiling, which have to be tuned to find on the device.

Heuristics for efficient hardware usage. We therefore formulate the hardware usage based on the tile sizes, and set heuristics to pick the ones with potential efficient hardware usage and good performance, as shown in Table 1. These will be the kernel optimization space for our JIT compilation, and evaluated interleaved with the DNN inferences on client devices. These formulas only need to get the client device type to know the hardware limitations of memory size, register size, and number of cores. This device type is supported to read by Web programming interface. No other prior knowledge of the devices are needed.

From the calculation, the ones priorities the higher-level storage (*i.e.,* registers) usage (Web GPU heuristics 4) within the storage size (Heuristics for Wasm 2 and Heuristics for WebGPU 2) will be in the space. They will be evaluated in the order from the ones with max activated blocks to the ones with min activated blocks (Web GPU heuristics 4). Other heuristics are based on the Wasm and WebGPU specification. Note in the actual calculation, there will be a relaxation

Table 1: Heuristics of web-specific lite space.

Params. (Symbols follow Fig. 10)	$DataBits$: bits of data, $x0$, $r0$, $y0$: the inner-most tile size, $x1$, $r1$, $y1$: the second tile size, $WarpSize$: the warp size, $Register_{core}$: the register number per core $L1Cache_{core}$: the L1 cache size per core, $Warp_{core}$: the warp number per core.
$SIMDLength$	$x0 \cdot DataBits$
$Thread_{core}$	$(x1/x0) \cdot (y1/y0)$
$Register_{thread}$	$((x0 \cdot r0) + (r0 \cdot y0) + (x0 \cdot y0))$
L1Cache/shared mem.	$(x1 \cdot r1) + (r1 \cdot y1) + (x1 \cdot y1)$
$Warp$	$Thread_{core}/WarpSize$
$ActiveBlock$	min(RegLimitedBlock, L1CacheLimitedBlock, warpLimitedBlock), $RegLimitedBlock = \frac{Reg_{core}}{Reg_{thread} \cdot Thread_{core}}$, $L1CacheLimitedBlock = \frac{L1Cache_{core}}{L1Cache}$, $warpLimitedBlock = \frac{Warp_{core}}{Warp}$
Heuristics for all	1. $SIMDlength \leq 128bits$ 2. $x0, r0, y0, x1, r1, y1$: power of 2
Heuristics for WASM	1. $L1Cache \leq L1Cache_{core}$ 2. $Register_{thread} \cdot Thread_{core} = Register_{core}$
Heuristics for WebGPU	1. $Thread_{core} \leq 256$ 2. $Register_{thread} \cdot Thread_{core} \leq Register_{core}$ 3. $L1Cache \leq 16KiB$ 4. $ActiveBlock = RegisterLimitedBlock$ 5. $ActiveBlock$: from max to min

ratio for the hardware resources, since other variables in the kernel will also use registers.

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.

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.

Major Primitives	Configures (symbols follow Table 1)
Cache Read	[Yes,No]
Cache Write	Yes
Reorder	$r2, y2, x2, r1, y1, x1,$ $r0, y0, x0$
Bind	$y2 \rightarrow$ block.y, $x2 \rightarrow$ block.x $y1 \rightarrow$ thread.y, $x1 \rightarrow$ thread.x
Unroll	$y0$
Vectorize	$x0$
Tile Size	$(r0, y0, x0) \in [4, ..., 32]$ $(r1, y1, x1) \in [32, ..., 256]$

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.

6 IMPLEMENTATION

The implementation of nn-JIT.web is based on TVM [10] and Binaryen [9]. We use the native GPU driver to compile WebGPU. To implement the Tensor-Web co-designed compilation pipeline in nn-JIT.web, we introduce a new compilation target for Wasm in TVM. Specifically, 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.

To create the lite kernel space for nn-JIT.web, 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. To implement the microbenchmark suite, we use a 4 k×4 k×4 k MatMul with different primitive settings to develop the evaluated kernels and compile them using the compilation tool chain of nn-JIT.web. We also adapt the in-browser inference runtime based on TVM, which enables importing graph definitions and weights from both TF.js and Ort-Web. Overall, nn-JIT.web comprises 2085 new lines of Python code, 1671 new lines of C++ code, and 564 new lines of JavaScript code.

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

7 EVALUATION

7.1 Experiment Setup

Hardware. We conduct experiments on 8 desktop and mobile devices, including AMD Ryzen 5800H CPU, Intel I9-12900 CPU, ARM Cortex-A78/A76 CPU and NVIDIA RTX 3000/3070 Ti GPU, AMD Radeon GPU, Intel HD 630 GPU. We fix the maximum frequency on the selected devices to ensure consistent performance measurements.

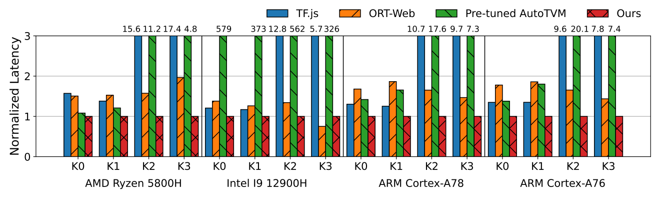
Kernels and models. We evaluate nn-JIT.web on modern transformer models, including RoBERTa [22], BART [20], GPT-2 [26], and T5 [27]. The kernel evaluation results show the typical sized kernels from these models, as listed in Table 3 including MatMul and BatchMatMul with different shapes. For the sequence-to-sequence models, such as GPT-2 and T5, we fix the input length at 384 to obtain the comparable results across devices and models.

Baselines. We compare nn-JIT.web with three in-browser DL inference frameworks as baselines, including TF.js, ORT-web and pre-tuned AutoTVM [10]. For TF.js, we use 3.21.0 version. For ORT-web, we use 1.14.0 version. For AutoTVM, we use the default kernel space, search algorithm *i.e.,* XG-Boost and tuning trails *i.e.,* 1000 to generate and tune the kernels. The turning targets are Intel I7-10700 CPU and WebGPU kernels on NVIDIA 3050 GPU, which are not included in our test devices. The evaluation is conducted in a Chrome browser with the version of 111.0.5555.

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 executions, the averaged latency is reported.

7.2 Overall Performance

Kernel Performance. Fig. 11 demostrates the latency of tested kernels on selected CPUs and GPUs, comparing baselines with nn-JIT.web. We observe a significant speedup. On CPUs with Wasm, nn-JIT.web achieves an average speedup of 42.57×, and on GPUs with WebGPU, it accelerates kernel executions by an average of 2.77×. Specifically, nn-JIT.web


(a) WASM

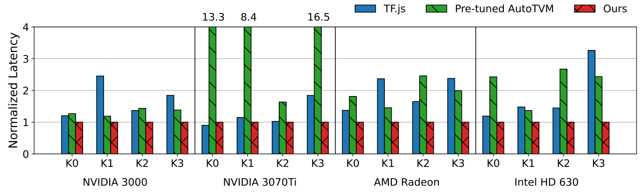
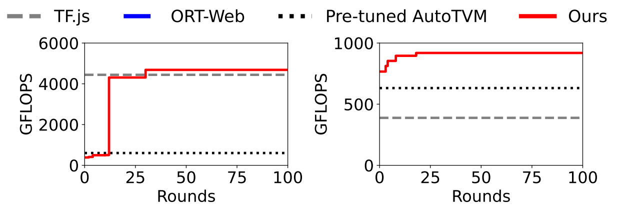

(b) WebGPU

Figure 11: Kernel latency executed with TF.js, ORT-web, pre-tuned AutoTVM as well as nn-JIT.web.


(a) NVIDIA 3070Ti (b) AMD Radeon

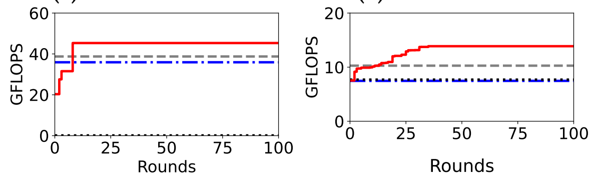

(c) Intel I9 12900H (d) ARM Cortex-A76

Figure 12: Kernel performance improvements with the JIT kernel optimization rounds on different devices.

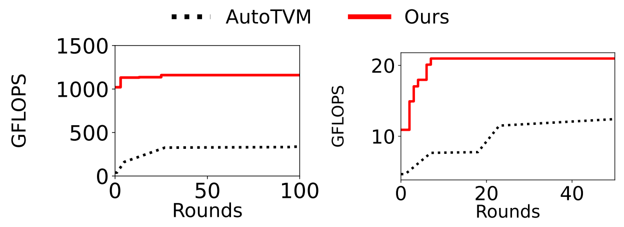

(a) NVIDIA 3050 (b) Intel I7 10700

Figure 13: Kernel performance improvements along with the JIT tuning rounds with nn-JIT.web and AutoTVM.

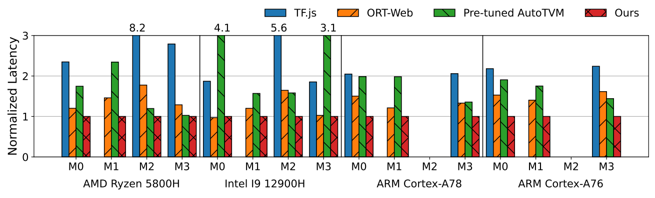
outperforms TF.js by 6.26× on Wasm and 1.68× on WebGPU. When compared to pre-tuned AutoTVM, the speedup is 119.92× on CPUs and 3.86× on GPUs. The inference speedup of nn-JIT.web is mainly due to efficient kernel tuning for specific hardware, whereas *one-for-all* kernel approaches including TF.js, ORT-Web as well as pre-tuned AutoTVM, fall short in this regard.

Figure12 showcases the kernel performance in GFLOPs and the JIT tuning rounds on selected CPUs and GPUs. We use the K2 kernel configuration, as detailed in Table3. As shown, nn-JIT.web attains optimal performance on CPUs with Wasm after 10~32 tuning rounds, while 25~40 rounds are needed to achieve peak performance on GPUs. This can be attributed to the web-specific lite space. Moreover, our compilation pipeline optimization ensures that each tuning round takes only about 500ms for Wasm and 100ms for WebGPU, based on our evaluation.

We also compare nn-JIT.web with the SOTA *one-for-each* kernel approach. We use AutoTVM to generate kernel candidates and perform JIT inference as JIT TVM. The kernel configuration employed is K0, as described in Table3. The comparison is illustrated in Figure13. As shown, on the Nvidia 3050, nn-JIT.web reaches near-peak performance (1159 GFLOPs) within 25 rounds, while AutoTVM lags behind at 350 GFLOPs. On the Intel I7, nn-JIT.web finds the best kernel implementation at the 8th round, demonstrating a significant performance improvement of approximately 2.80× compared to AutoTVM. Our evaluation suggests that AutoTVM would need 1106 additional tuning rounds to achieve its optimal performance.

Model performance. We continue to evaluate the end-to-end model performance achieved by nn-JIT.web and other baselines. In Figure 14, we denote RoBERTa as M0, BART as M1, GPT-2 as M2, and T5 as M3. As illustrated, nn-JIT.web attains more than 3.13× and 1.36× 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, nn-JIT.web improves by up to 8.27× on M3 compared to TF.js, while up to 5.64× on Intel I9. Compared to pre-tuned AutoTVM, the achieved speedup is approximately 1.93× with Wasm and 1.51× with WebGPU. As TF.js and ORT-Web cannot support all kernels in the tested models with WebGPU, we do not report their model latency.

The JIT kernel optimization on models may take longer than on individual kernels, but it remains efficient. As shown in Figure 15, for BART, nn-JIT.web takes approximately 5.5 seconds to discover the optimal kernel implementations for the Nvidia 3000 GPU. For CPUs, peak performance is just achieved after around 17.8s and 22.1s for the Intel i9 and ARM A76, respectively.


(a) WASM

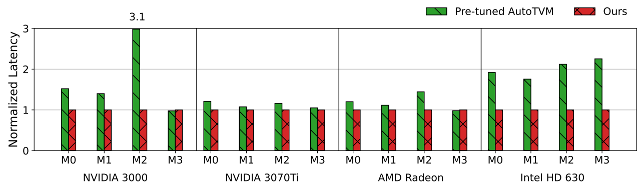
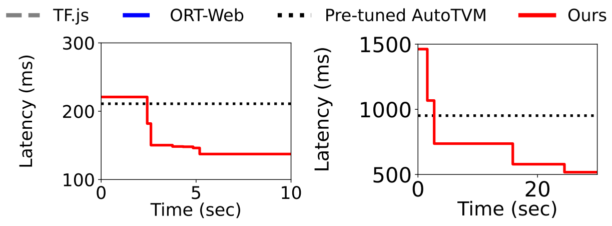

(b) WebGPU

Figure 14: Model latency executed with TF.js, ORT-web, pre-tuned AutoTVM as well as nn-JIT.web.


(a) NVIDIA 3000 (b) Intel HD 630

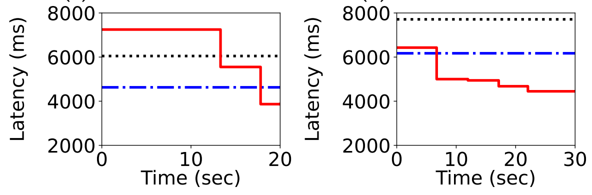

(c) Intel I9 12900H (d) ARM Cortex-A76

Figure 15: Model performance improvements with the JIT kernel optimization time on different devices.

7.3 Evaluation of Tensor-Web Co-Designed Compilation Pipeline

Next, we analyze nn-JIT.web and evaluate each design component, beginning with the optimized compilation pipeline. Table 4 presents the compiling latency and achieved kernel latency for both the baseline and nn-JIT.web 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 pass, combined instruction pass, and load/store to variable, to assess their individual contributions to the optimized pipeline. 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%.

Table 4: The compiling latency and achieved kernel latency of nn-JIT.web with different optimization passes.

	Compilation Latency (sec)	Inference 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

7.4 Evaluation of Web-Specific Lite Space

Next, we assess the design of the web-specific lite kernel space, which significantly reduces the kernel optimization space. We examine two typical kernels, MatMul and BatchMatMul, and compare their kernel optimization spaces in AutoTVM and nn-JIT.web. The results are presented in Table 5. 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, nn-JIT.web reduces the overall kernel generation cost from hours to milliseconds, enabling JIT-powered inference in web browsers.

Table 5: Kernel space of AutoTVM and nn-JIT.web.

Kernel Type (Size)	AutoTVM		nn-JIT.web	
	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

7.5 Overhead

nn-JIT.web 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 clients. The compiled kernel sizes range between 5~30KiB, which does not add a significant burden to the network load. To evaluate the newly arriving kernels, the client typically takes around 69~728ms for most kernels based on our assessment, which is nearly imperceptible.

8 RELATED WORKS

DL kernel generation. Many works focus on automatically searching and generating optimal kernel implementations from a vast space. TVM [10] generates DL kernels based on the space of manual schedule templates and a learned cost model to search for the best kernel implementation. The subsequent work, Ansor [34], generates higher-performance DL kernels than TVM without manual schedule templates and reduces the average search time. Romou [21] supports new primitives to generate mobile-GPU-friendly DL kernels and accelerates kernel generation through hardware-aware search space pruning. Roller [36] generates DL kernels using an rTile-construction-based approach, significantly reducing search time. Triton [30] is a DL kernel generator that extends LLVM-IR and adds an additional tile-level optimization pass, achieving high DL kernel performance. TLP [33] is a DL-based cost model that leverages schedule primitive features to speed up DL kernel search. However, none of these works address the online optimal kernel generation issue for in-browser DL inference. They all fail to provide the lightweight kernel space and compilation pipeline, which meet the requirements of JIT inference on the Web.

In-Browser DL inference. The emergence of DL frameworks, such as TensorFlow.js [29] and ONNX Runtime Web [32], has significantly contributed to making in-browser DL inference a reality. TensorFlow.js, proposed by Google, is an open-source library that enables the deployment of DL models in browsers or on Node.js. It supports JavaScript, Wasm, WebGL, and WebGPU. ONNX Runtime Web, another open-source library proposed by Microsoft, facilitates in-browser DL inference by processing models in ONNX format. However, it only supports Wasm and WebGL backends.

9 CONCLUSION

In this paper, we present nn-JIT.web, the first in-browser inference system that enables JIT optimized kernel generation, supporting Wasm and WebGPU. Our evaluation shows that nn-JIT.web accelerates inference by an average of 26.65× 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. ChatGPT. https://chat.openai.com/.

[3] Retrived in June, 2023. Microsoft 365. https://www.office.com/.

[4] Retrived in June, 2023. Stable Diffusion Online. https://stablediffusionweb.com/.

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

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

[20] 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

[21] 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

[22] 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

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

[24] 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

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

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

[27] 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

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

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

[30] 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

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

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

[33] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. TLP: A Deep Learning-Based Cost Model for Tensor Program Tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 833–845. https://doi.org/10.1145/3575693.3575737

[34] 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.

[35] 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

[36] 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