SUPPLEMENTARY FILE

Abstract Deep learning technique is drawing more and more attention to Web developers. A lot of Web apps perform inference of deep neural network (DNN) models within Web browsers to provide intelligent services for their users. Typically, GPU acceleration is required during DNN inference, especially on end devices. However, it has been revealed that GPU acceleration in Web browsers has an unacceptably long warm-up time, harming the quality of service (QoS) of DNN inference in Web apps. In this paper, we first analyze the reason for the long warm-up time, and find out that compiling the WebGL programs of DNN inference into binaries is the key bottleneck. Then we propose WPIA, an approach which enables Web apps to load binaries of precompiled WebGL programs rather than compiling on the fly, accelerating the DNN warm-up in Web browsers. WPIA consists of two phases. In the offline phase, WPIA collects the binaries of compiled WebGL programs, and merges similar WebGL programs to trim the size of the binaries as well as reduce the overhead of fetching and loading the binaries. In the online phase, WPIA loads the binaries into the users' Web browsers, using the record-and-replay technique to ensure the correctness of executing precompiled WebGL programs. Evaluation results show that WPIA can reduce 84.1% of the DNN warm-up time on average and 95.3% at maximum, accelerating DNN warm-up to an order of magnitude faster, with negligible additional overhead.

Keywords DNN warm-up, WebGL programs, precompiling

1 Introduction

With the advent of artificial intelligence, it is becoming increasingly common to leverage deep learning technologies to provide intelligent services in Web apps [1] [2]. Traditionally, Web apps perform deep neural network (DNN) inferences at cloud servers. With the increasing demand for real-time responses and privacy protection, an increasing number of Web apps have shifted DNN inferences to Web browsers. Compared with traditional approaches where DNN runs at cloud servers, DNN inference in Web browsers avoids server connections, boosting the responsiveness and enhancing user privacy of Web apps. DNN inference in Web browsers has been adopted by Web apps in a variety of fields, including Web conferencing (e.g., Google Meet [3] [1] and Amazon Chime [4] [2]), video chatting (e.g., Snapchat [5]) and graphics editing (e.g., Adobe Photoshop [6] [7]). Recently, large-scale models such as stable diffusion models [8] and large language models (LLM) [9] have also been ported to Web browsers.

Many efforts have been made to accelerate DNN model inference in Web browsers. On the one hand, GPU-related APIs such as WebGL [10] and experimental WebGPU [11] have been introduced into Web browsers. These APIs make it possible to execute heavy computing tasks for Web apps with GPU acceleration. On the other hand, DNN frameworks such as TensorFlow.js [12], ONNX Runtime Web [13] and WebDNN [14] have emerged. Almost all these DNN frameworks provide a *WebGL* backend, which leverages the WebGL APIs to implement GPU acceleration for DNN inference. These frameworks help Web apps perform DNN inference within Web browsers conveniently.

However, although using GPU for DNN inference has been shown to have better performance than using pure CPU backend [15] [16], our prior work has discovered that DNN inference on WebGL has an unacceptably long *warm-up* time [15]. Here *DNN warm-up* refers to the first inference after the DNN model is fully downloaded, where necessary preparation steps for DNN inference on WebGL take place. Since Web users, on average, spend less than a minute on a Web app [17], the loading time in Web apps is crucial to their user experience [18] [19]. The long warm-up time prevents Web apps from serving their users as soon as possible, which negatively impacts the quality of service (QoS) of Web apps [20].

To address the issue, in this paper, we analyze the reason for the long DNN warm-up time in Web browsers and find that compiling the WebGL programs used in DNN inference into binaries takes most of the time. To reduce the compiling time, we propose *WPIA* (WebGL Program Immediately Available), an approach which analyzes and precompiles WebGL programs into binaries at the server side to accelerate DNN warm-up in Web browsers. WPIA consists of an offline phase and an online phase. In the offline phase, WPIA analyzes and precompiles the WebGL programs needed in DNN inference at the server side. In the online phase, WPIA loads the binaries and arranges the execution of WebGL programs. The design of WPIA tackles two challenges in implementing the idea of precompilation. First, downloading binaries brings additional overhead to both time and bandwidth consumption in DNN warm-up. WPIA reduces the overhead by eliminating the shared code through program analysis. Second, precompiling changes the way of WebGL compilation from just-in-time into ahead-of-time, which raises problems in execution of WebGL programs in DNN inference. WPIA tackles this challenge by record-and-replay, which records the execution order of WebGL programs in the offline phase and replays it in the online phase. WPIA's record-and-replay supports both DNN warm-up and later DNN inference.

We implement a prototype of WPIA based on two open-

source projects: Chromium [21] and TensorFlow.js [12]. We pierce through the entire GPU software stack of Chromium and expose necessary APIs for WPIA to load binaries of compiled WebGL programs. Based on the prototype, we evaluate the performance and overhead of WPIA on different devices and DNN models. The results show that WPIA can reduce 84.1% of total DNN warm-up time on average and 95.3% maximally, which is one order of magnitude faster, with only negligible overhead in loading precompiled programs.

In summary, we make the following main contributions in this paper:

- We investigate the reason for the long DNN model warm-up time in Web apps and find that compiling WebGL programs into binaries takes most of the time.
- We propose WPIA, an approach which reduces the DNN warm-up time in Web apps by precompiling WebGL programs offline.
- We evaluate WPIA, and results show that WPIA can effectively reduce the DNN warm-up time in Web browsers with negligible overhead.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation of this paper. Section 3 elaborates on the design of WPIA. Section 4 presents the prototype implementation of WPIA. Section 5 analyzes the evaluation results of WPIA. Section 6 discusses some possible issues when deploying WPIA in practice. Section 7 summarizes related work, and Section 8 concludes this paper.

2 Background and Motivation

In this section, we will briefly introduce GPU support in Web browsers and DNN inference in Web apps. We will then reveal the bottleneck of DNN warm-up in Web browsers.

2.1 GPU support in Web Browsers

Modern Web browsers provide the WebGL APIs [10] to support GPU acceleration for Web apps. With WebGL APIs, Web apps can submit graphics computing tasks to the local GPU, significantly reducing the rendering latency. Each WebGL context is associated with an HTML Canvas element [22]. For a WebGL context, the WebGL calls are executed sequentially one by one. No two WebGL calls will be executed in parallel or concurrently. This design follows the principle of OpenGL ES APIs, from which WebGL APIs are derived. The sequential execution model of WebGL APIs guarantees a synchronous execution environment for Web apps, in accordance with the main programming language, the single-threaded JavaScript [23].

Web apps customize the rendering behavior using *WebGL shaders*. A WebGL shader is a snippet of code composed by a Web app and defines some rendering computation on a local GPU. WebGL shaders are written in a C-like language called Graphics Library Shader Language (GLSL). All

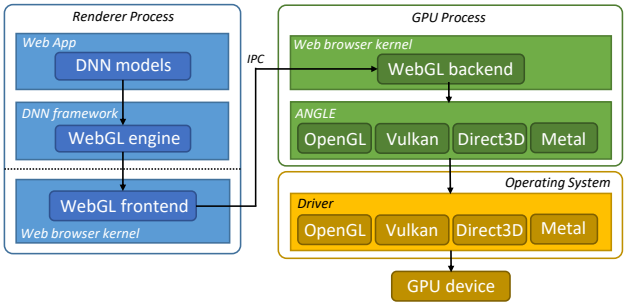


Fig. 1: The GPU stack in Chrome

WebGL shaders needed by a whole rendering pipeline form a *WebGL program*. Before execution, a WebGL program must be compiled and linked into a binary, which is then sent to the GPU and executed by it. The WebGL specification does not standardize the format of the binary, and the format is entirely dependent on the models of GPU devices.

Web browsers implement the WebGL APIs for Web apps based on the native GPU support of the underlying operating systems. As an example, the GPU stack of Chromium is shown in Figure 1. Chromium adopts a multi-process architecture to implement WebGL support. The WebGL frontend in a renderer process receives calls from Web apps and sends the calls to the WebGL backend in the GPU process through inter-process communication (IPC). Although WebGL calls can be trivially converted into OpenGL calls, in practice, because of performance concerns of OpenGL, mainstream Web browsers use the ANGLE project [24] to translate WebGL into modern GPU APIs such as Vulkan, Direct3D and Metal on different operating systems. Therefore, WebGL calls from Web apps are converted into calls to native GPU APIs and then handed over to the operating systems. Finally, GPU drivers in the OSs interpret the calls and control the GPU hardware to execute the GPU tasks. Execution results obtained from native GPU APIs are wrapped by Web browsers and returned to Web apps.

2.2 DNN Inference in Web Apps

DNN inference is the process of computing the outputs of a DNN model given its inputs. A DNN model consists of a computational graph and the model weights. The computational graph is a directed graph that represents the computational steps that the DNN model performs. It is made up of *DNN operators* (Ops), which are the basic building blocks of the graph. Each operator performs a specific computation, taking some *tensors* as inputs and generating some tensors as outputs. When a DNN model is used for inference, the input data is passed to the computational graph. The graph then performs the computations specified by the operators, using the model weights to configure the computations. The outputs of the graph are the predictions of the DNN model.

GPU acceleration is often required to perform DNN infer-

ence in users' devices [25] [26]. DNN frameworks leverage the WebGL APIs in Web browsers to implement GPU acceleration. We call an inference engine based on the WebGL APIs as the *WebGL backend*. A WebGL backend stores tensors as *WebGL textures* and implements the computation of operators as *WebGL programs*.

A DNN framework breaks down a DNN inference task into executing a sequence of DNN operators based on the structure of the given DNN model. A WebGL backend in the framework handles the execution of DNN operators. Before executing a DNN operator, the WebGL backend composes the source code of the needed WebGL programs and compiles them into binaries on the fly. If necessary, the WebGL backend then uploads the operator's inputs to the GPU. Finally, the WebGL backend executes the binaries of compiled WebGL programs on the uploaded input data, which finishes the computation of a DNN operator.

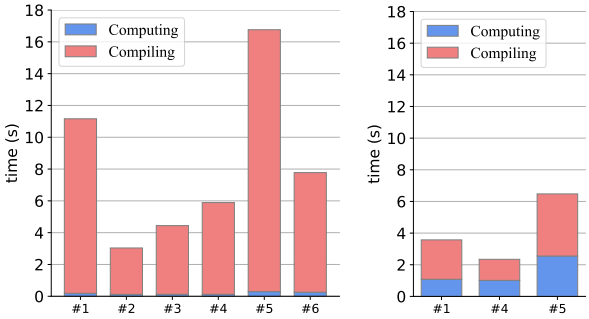
2.3 DNN Warm-up in Web browsers

After downloading a DNN model, Web app should execute DNN inference for once to make the WebGL backend well prepared for later inference. In this paper, we denote the period between the time when the DNN models are fully downloaded and the time when the first model inference completes as *DNN warm-up*, and denote the time elapsed between DNN warm-up as *DNN warm-up time*.

Our prior study [15] has revealed that DNN inference in Web browsers has a long warm-up time. Based on our prior work, we take a step further and analyze the reason behind this phenomenon. We find out that this phenomenon is caused by two main tasks that take place during the warm-up:

1. Compiling WebGL programs used in DNN inference into binaries;
2. Executing computation of DNN inference.

We measure the time that these tasks take up in the DNN warm-up in different situations. Specifically, we conduct experiments on a Hasee laptop with Windows 10, with two different DNN frameworks: TensorFlow.js [12] and WebDNN [14]. TensorFlow.js is released by Google, and it can support a diverse spectrum of deep learning tasks and DNN models. Researchers at Tokyo University develop WebDNN, and they claim [27] that WebDNN is the fastest DNN execution framework on Web browsers. We believe these two frameworks can represent current best practices in industrial and research communities, respectively. We use six DNN models: GoogleNet [28] (#1), MobileNetV1 [29] (#2), MobileNetV2 [30] (#3), SqueezeNet [31] (#4), DenseNet [32] (#5), and EfficientNetV2 [33] (#6). Since TensorFlow.js and WebDNN accept DNN models in different formats, to ensure the consistency of DNN inference tasks, we implement the models in the Keras format and then convert the models into different formats. In each environment, we perform a DNN warm-up and measure the time it takes to execute each type of the preceding tasks. WebDNN fails to execute models #2, #3, and #6 due to lack of implementation of some DNN operators in WebGL. We repeat each experiment three times and present the average results in Figure 2.



(a) TensorFlow.js (b) WebDNN

Fig. 2: Analysis of the long DNN warm-up time

Figure 2 shows that compiling WebGL programs into binaries takes most of the DNN warm-up time. For TensorFlow.js, it can take tens of seconds for the WebGL backend to compile the WebGL programs for most DNN models, whereas the DNN inference usually takes less than 0.3 seconds. Surprisingly, compiling WebGL programs takes more than 95% of the time in the DNN warm-up for all DNN models on TensorFlow.js. For WebDNN, compiling WebGL programs takes less time than TensorFlow.js, but it can still take several seconds, or more than 50% of the DNN warm-up time.

From the results, we know that reducing the time spent compiling WebGL programs into binaries is critical to effectively accelerating DNN warm-up in Web browsers. Therefore, We propose WPIA, an approach which precompiles the WebGL programs at remote servers to avoid compiling them on the fly during DNN warm-up.

3 Design

This section will introduce the design of WPIA. We will first provide an overview of our approach, then discuss the design challenges, and finally present the design details.

3.1 Overview

Figure 3 shows the overview of WPIA, which consists of two phases: an offline phase at the server side and an online phase at the Web browser side.

In the offline phase, WPIA collects the binaries of the WebGL programs used in DNN inference. Given a DNN model, WPIA collects necessary information on the WebGL programs that are compiled during the DNN warm-up by performing a DNN inference. During the inference, WPIA records the source code and the execution order of WebGL programs (Section 3.4). After the DNN inference finishes, WPIA merges all WebGL programs (Section 3.3) used in the

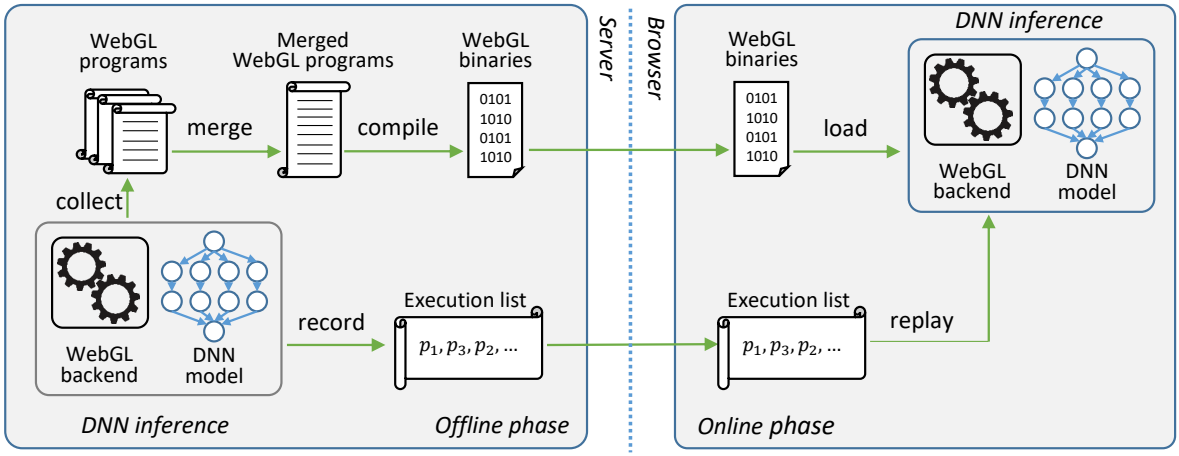


Fig. 3: Overview of WPIA

inference. Finally, WPIA compiles the WebGL programs into binaries (Section 3.2) and saves the binaries at the local storage.

The online phase begins when a user visits the Web app. WPIA first fetches the binaries of precompiled WebGL programs from the Web server and then loads the binaries into the Web browser. During warm-up, WPIA replays the execution order of WebGL programs recorded in the offline phase (Section 3.4). No WebGL programs are generated or compiled on-the-fly during the DNN inference. Therefore, WPIA avoids compiling WebGL programs into binaries and greatly reduces the DNN warm-up time in Web browsers.

Challenges. The design of WPIA faces two challenges.

1. *Reducing the overhead of loading precompiled binaries*. WPIA needs to fetch the binaries of precompiled WebGL programs and load them into Web browsers. This process can add overhead during DNN warm-up, so WPIA must find ways to reduce the size of the binaries.

2. *Arranging execution of WebGL programs during DNN inference*. WPIA changes the just-in-time compilation of WebGL programs to ahead-of-time compilation. This raises challenges in arranging the execution of perecompiled WebGL programs in DNN inference.

Next, we will describe WPIA in detail, including how it handles the challenges of reducing the total size of the binaries and selecting the correct binary during DNN inference.

3.2 Precompiling WebGL Programs

In the offline phase, WPIA performs WebGL program precompiling on the server side. First, WPIA simulates a client-side inference of the given DNN model at the server. During the inference, WPIA collects the source code of all WebGL programs generated by the WebGL backend. Then WPIA precompiles the source code and obtains the binaries of these WebGL programs. These binaries are saved in local storage and will later be delivered to users' Web browsers in the online phase.

The way WPIA precompiles WebGL programs is consistent with Web browsers. Generally, Web browsers convert WebGL programs into OpenGL programs and compile them using OpenGL libraries. WPIA compiles WebGL programs using the OpenGL libraries in the same way as Web browsers do, ensuring that the program binaries can work correctly when loaded into users' Web browsers.

GPU vendors specify the formats of WebGL program binaries and generally do not disclose the formats to the public, which means that the same WebGL program can have different compiling results on different devices. To enable WebGL program precompiling for more users, we should compile the programs on various devices on the server side. This approach is commonly used in research that optimizes client-side DNN inference [34]. In the online phase, WPIA first checks the GPU vendor of the user devices and fetches the corresponding precompiled WebGL programs from the servers.

3.3 Merging WebGL Programs

Loading precompiled WebGL programs from WPIA reduces DNN warm-up time but also introduces additional overhead in downloading the binaries. The larger the binaries, the more time and network bandwidth will be consumed by Web browsers.

We study the WebGL programs used during DNN inference and find that a large proportion of the source code is duplicated between different WebGL programs. To unveil the reason for code duplication among WebGL programs, we collect all 69 WebGL programs used to infer a DNN model named Iris. Iris is the most downloaded DNN model on TFHub [35], which is the official website of Google for publishing DNN models. Figure 4a briefly summarizes two programs used in the inference of Iris. These two programs both implement matrix multiplication with WebGL, whereas they accept different sizes of matrices as their inputs. We compare the source code of these two programs. We find that they

Line No.	Source code summary of matrix multiplication in WebGL	
1	`#version 300 es`	
	...	
69	`float random(float seed){`	*common math functions*
77	`vec2 uvFromFlat(int texNumR, int texNumC, int index){`	
	...	
126	`void setOutput(vec4 val){`	*encoding and decoding textures*
130	`uniform sampler2D matrixA;`	
	...	
135	`uniform int offsetbias;`	*uniform variables*
137	`ivec3 getOutputCoords(){`	
153	`vec4 getMatrixA(int row, int col){`	
	...	
196	`vec4 getBiasAtOutCoords(){`	*locating elements in matrics*
206	`vec4 activation(vec4 x){`	
	...	
228	`void main(){`	*calculation of matrix multiplication*

(a) Function list of the two programs. Identical functions are marked in green and others (only the function `getOutputCoords()` and `getMatrixA()`) are marked in red.

```
ivec3 getOutputCoords() {
  ivec2 resTexRC = ivec2(resultUV.yx *
                   vec2(2, 64));
  int index = resTexRC.x * 64 +
              resTexRC.y;
  int b = index / 128;
  index -= b * 128;
  int r = 2 * (index / 64);
  int c = imod(index, 64) * 2;
  return ivec3(b, r, c);
}
```

```
ivec3 getOutputCoords() {
  ivec2 resTexRC = ivec2(resultUV.yx *
                   vec2(32, 64));
  int index = resTexRC.x * 64 +
              resTexRC.y;
  int b = index / 2048;
  index -= b * 2048;
  int r = 2 * (index / 64);
  int c = imod(index, 64) * 2;
  return ivec3(b, r, c);
}
```

```
vec4 getMatrixA(int row, int col) {
  vec2 uv = (vec2(col, row) + halfCR) /
            vec2(64.0, 4.0);
  return texture(matrixA, uv);
}
```

```
vec4 getMatrixA(int row, int col) {
  vec2 uv = (vec2(col, row) + halfCR) /
            vec2(64.0, 64.0);
  return texture(matrixA, uv);
}
```

(b) Source code of `getOutputCoords()` and `getMatrixA()`. Code differ between the two programs is marked in red.

Fig. 4: Comparison of two programs in DNN inference

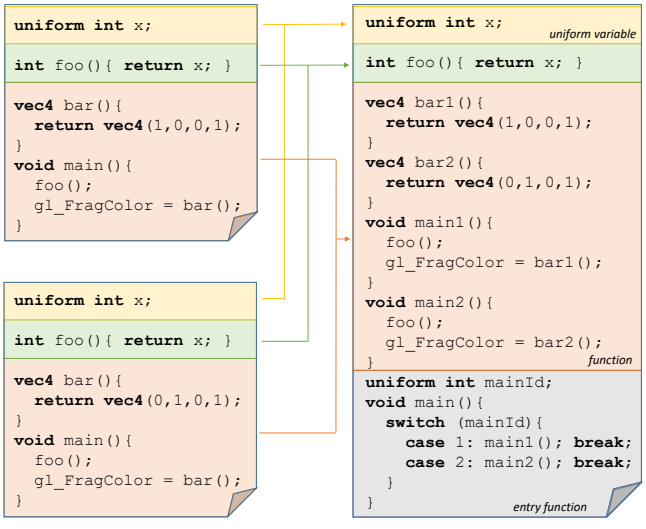


Fig. 5: Merging of two WebGL programs

both have 27 functions and 238 lines of code. Among them, 25 functions (92.5%) and 234 (98.3%) lines of source code are duplicated. Only two functions are different between the two programs, and the differences between these two functions lie only in some constant numbers, as shown in Figure 4b. The duplicated code includes helper functions for common math operations, encoding and decoding textures, and locating elements in matrices. Because WebGL programs are standalone and cannot be linked with or call other programs, DNN frameworks must include these helper functions in source code of all WebGL programs, leading to serious code duplication.

WPIA removes duplicated code by program merging, in which WPIA merges similar WebGL programs and keeps only one copy of the duplicated code in the merged WebGL program.

Diving into the structure of WebGL programs used in DNN inference, we find that these WebGL programs can be divided into the following code snippets: *uniform variables*, *functions*, *global variables*, and an *entry function*. Therefore, WPIA expresses a WebGL program as a quadruple $P = (U, F, V, e)$, where U, F, V, e corresponds to the set of uniform variables, the set of functions, the set of global variables, and the entry function, respectively. Before WPIA merges WebGL programs, it first analyzes the source code and extracts U, F, V, e. WPIA then merges these code snippets one by one.

- *Extracting uniform variables.*

WebGL programs declare special global variables called *uniform variables* to receive inputs from JavaScript code. Uniform variables are fed with values by JavaScript code and act as global variables in WebGL programs.

We study the characteristics of uniform variables in WebGL programs for DNN inference and find that the number of uniform variables is highly related to the number of inputs that DNN operators need. For example, an activation operator such as Sigmoid and ReLu accepts one matrix as its input, so the WebGL program for the operator tends to have one corresponding uniform variable. Programs for binary operators, such as element-wise addition, have two corresponding uniform variables for the two input matrices. If two DNN operators accept the same number of inputs, their WebGL programs tend to have the same set of uniform variables. Based on this finding, WPIA categorizes WebGL programs according to their set of uniform variables. WebGL programs that have the same set of uniform variables are categorized into the same group. The programs in a group are then merged into a single WebGL program. WPIA's uniform-based categorization of WebGL programs greatly reduced the number of programs after merging while maintaining the runtime correctness of the programs.

- *Merging functions and global variables.*

Merging functions and variables are the kernel of WPIA's merging of WebGL programs. Figure 5 briefly illustrates how

Table 1: Properties used in program merging of WPIA

Property	Description	Example
f.name	Function name	`func`
f.sig	Function signature	`int func(int)`
f.body	Function body	`{return 0;}`
v.name	Global variable name	`var`
v.sig	Global variable signature	`int var`

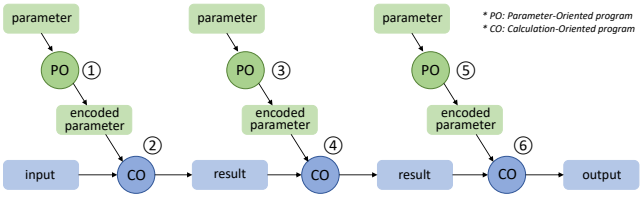
Algorithm 1 Merging two WebGL programs

Require: $P = (U, V, F, e), P' = (U, V', F', e')$

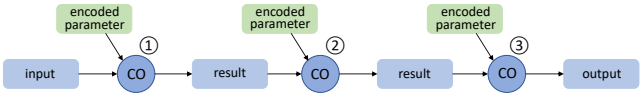
1: $\mathbb{V} \leftarrow \phi; \mathbb{F} \leftarrow \phi$
2: **for** $v \in V \cup V'$ **do**
3: **if** $\exists \bar{v} \in \mathbb{V}$, s.t. v.sig = $\bar{v}$.sig **then**
4: v.name $\leftarrow \bar{v}$.name; **continue;**
5: **end if**
6: $\mathbb{V} \leftarrow \mathbb{V} \cup \{v\}$; v.name $\leftarrow$ newUniqueName()
7: **end for**
8: **for** $f \in F \cup F'$ **do**
9: f.body $\leftarrow$ mapNameToNewNames(f.body)
10: **if** $\exists \bar{f} \in \mathbb{F}$, s.t. f.sig = $\bar{f}$.sig $\wedge$ f.body = $\bar{f}$.body **then**
11: f.name $\leftarrow \bar{f}$.name; **continue;**
12: **end if**
13: $\mathbb{F} \leftarrow \mathbb{F} \cup \{f\}$; f.name $\leftarrow$ newUniqueName()
14: **end for**
15: $\bar{U} \leftarrow U \cup \{\text{mainId}\}$; $\bar{e} \leftarrow$ genEntryFunction()
16: **return** $(\bar{U}, \mathbb{V}, \mathbb{F}, \bar{e})$

WPIA merges two WebGL programs. Let us denote a global variable as v and a WebGL function as f. The properties of v and f needed by WPIA in merging are listed in Table 1.

Algorithm 1 describes WPIA's merging algorithm. From Line 1 ~ 7, the pseudocode shows how WPIA merges global variables from different programs. WPIA checks whether a variable is declared in the other program one by one (Line 2). For some variable v, if a variable $\bar{v}$ with the same signature is found (Line 3), i.e., there is another variable with the same name and type in other programs, the two variables will be merged (Line 4); otherwise, the variable v will be assigned with a unique name (Line 6). The merging of functions (Line 8 ~ 14) in WPIA adopts a similar approach, with an additional step of mapping global variables used in the functions into their new names. WPIA merges WebGL functions based on their function signatures and bodies. For any global variables and other functions that are referred to in the body of a function f, WPIA changes their names to the new names assigned by WPIA during merging (Line 9). This step avoids potential conflicts of function calling. As we can see in Figure 5, the two `main()` functions from the two programs are completely identical in their code, but they have different behaviour and should not be merged together. WPIA iterates over functions following their topological orders in the function call graph. If two functions with the same signatures have the same function body (Line 10), it means that these



(a) The execution order of WebGL programs in DNN warm-up



(b) The execution order of WebGL programs in later DNN inference.

Fig. 6: Execution of WebGL programs in DNN inference.

two functions are identical, so WPIA only keeps one copy of the function (Line 11); otherwise, WPIA assign a new name to the function (Line 13). We can see that WPIA's merging of WebGL programs adopts a simple code clone detection [36] [37] algorithm in it. The simple detection is effective in detecting the code clone against generated WebGL programs in DNN inference while guaranteeing the soundness and correctness of program merging.

- *Generating the entry function.*

After WPIA merges several WebGL programs, the original entry functions `main()` will be assigned with different function IDs, losing the identities of entry functions. WPIA then generates a new `main()` entry function as well as a uniform variable named `mainId` (Line 15 in Algorithm 1). If WPIA needs to execute an original WebGL program from a merged WebGL program, WPIA will pass the corresponding function ID to the `mainId` uniform variable. The new `main()` function will call the correct original entry function according to the `mainId`.

WPIA's merging algorithm resolves potential name conflicts between different WebGL programs, which ensures that WPIA can execute the merged WebGL programs coherently and correctly.

3.4 Executing Precompiled WebGL Programs

Precompiling in WPIA reduces the compiling time but also raises new problems in executing DNN inference. Originally, WebGL programs are generated and compiled just-in-time during DNN inference. Since WPIA precompiles and directly loads the WebGL programs into Web browsers, WPIA needs to correctly arrange the execution of precompiled WebGL programs with the execution of DNN operators.

WPIA uses a *record-and-replay* technique to tackle this problem. Because of the single-threaded nature of JavaScript and WebGL, it is possible to record the order in the offline phase and replay it in the online phase.

By studying the execution of WebGL programs in DNN inference, we find that the WebGL programs have two dif-

ferent kinds of functionalities. We illustrate the execution of WebGL programs in DNN inference in Figure 6. From Figure 6, we can see that WebGL programs in the DNN inference can be classified into two categories: *Parameter-Oriented (PO)* WebGL programs and *Calculation-Oriented (CO)* WebGL programs. PO programs are applied to the parameters (weights) of the DNN model, while CO programs are fed with DNN inputs or intermediate results. For example, WebGL programs that encode parameter matrices into WebGL textures are typical PO programs. During the DNN warm-up, all PO programs and CO programs will be executed. But during later DNN inference, only CO programs will be executed again on new DNN inputs because the execution results of PO programs are consistent and can be reused.

According to the different usage of PO and CO programs, WPIA uses separate approaches to handle these two kinds of WebGL programs in record-and-replay. In the offline phase, WPIA records the execution order of WebGL programs in the DNN warm-up. The recorded execution sequence includes all CO programs and PO programs. Then, WPIA performs the DNN inference for the second time. The second execution sequence only includes all CO programs. In the online phase, during the DNN warm-up, WPIA replays the first recorded execution list. In later DNN inference, WPIA replays the second recorded execution list.

4 Implementation

We implement a prototype of WPIA based on several large open-source projects. Our prototype consists of an instrumented Web browser and a modified DNN framework on Web platforms.

The instrumented Web browser is implemented based on a modern Web browser, Chromium 99.0.4795.1 [21]. We pierce through the GPU software stack in Chromium, providing APIs to extract the binary of a compiled WebGL program from the Web browser and load the binary into the browser.

In the offline phase, we use *TensorFlow.js 3.15.0* [12] (TF.js) to collect the information about WebGL programs in DNN inference. We make several modifications to TF.js according to our requirements. We add a new API in TF.js to expose WebGL programs used in the DNN inference. We also make the WebGL backend in the modified TF.js able to record the execution list of WebGL programs. Instead of using lexical parsers [38], we manually annotate the code templates in TF.js to help the toolkit to analyze the source code of WebGL programs.

In the online phase, the WPIA JS library is also implemented based on *TensorFlow.js 3.15.0*. We modify TF.js to use binaries of precompiled programs in the DNN inference instead of composing and compiling WebGL programs into binaries on the fly. Before the DNN warm-up, the WPIA JS library fetches the precompiled binaries of WebGL programs and loads them into the browser. During the DNN inference, the modified TF.js can handle the merged WebGL programs and correctly set the `mainId` uniform variable before executing a WebGL program. We also add the replaying functionality to the WebGL backend in the modified TF.js.

We have carefully checked the correctness of our prototype implementation. We verified the consistency between the inference results of the vanilla TF.js framework and WPIA under various DNN models and DNN inputs.

5 Evaluation

In this section, we will describe the evaluation experiments and measure the performance and overhead of WPIA.

5.1 Evaluation Setup

We comprehensively evaluate the performance of WPIA. The experiments are carried out on four devices and six DNN models. The details of the experiments are described as follows.

Devices. We use four devices in the experiments, including two laptops and two mobile phones. The devices in experiments can be viewed in Table 2. We use the Pixel 3 to represent high-end mobile phones and the Redmi Note 8 to represent low-end mobile phone.

Models. We use six well-known DNN models in the experiments: GoogLeNet [28], MobileNetV1 [29], MobileNetV2 [30], SqueezeNet [31], DenseNet [32], and EfficientNetV2 [33]. These DNN models are widely used in the end devices and in evaluating DNN inference frameworks [15] [25]. The detailed information of these DNN models can be viewed in Table 3.

Baselines. We compare the performance of WPIA with the original WebGL backend in TensorFlow.js, which we call the vanilla approach in this paper. In addition, we implement a WPIA$_r$ approach, which loads precompiled WebGL programs but does not merge the programs. The vanilla approach and WPIA$_r$ are used as the baseline approaches in our evaluation.

Research questions. We propose the following three research questions to evaluate the performance and overhead of WPIA.

1. How effective does WPIA reduce the DNN warm-up time?
2. How does the WebGL program merging contribute to WPIA?
3. What is the overhead of WPIA?

We will answer these questions in the following subsections.

5.2 RQ1: How Effective Does WPIA Reduce the DNN Warm-up Time?

To answer RQ1, we measure the DNN warm-up time of different approaches on all the aforementioned DNN models

Table 2: Device information

ID	Device	OS	CPU	GPU	Dedicated GPU	Memory
H	Hasee	Windows 10	Intel Core i7	Intel UHD Graphics	NVIDIA GeForce GTX 1070	8GB
D	Dell Inspiron	Windows 10	Intel Core i7	Intel Iris Xe Graphics	NVIDIA GeForce MX350	16GB
P	Pixel 3	Android 9	Snapdragon 845	Adreno 630	N/A	4GB
R	Redmi Note 8	Android 9	Snapdragon 665	Adreno 610	N/A	3GB

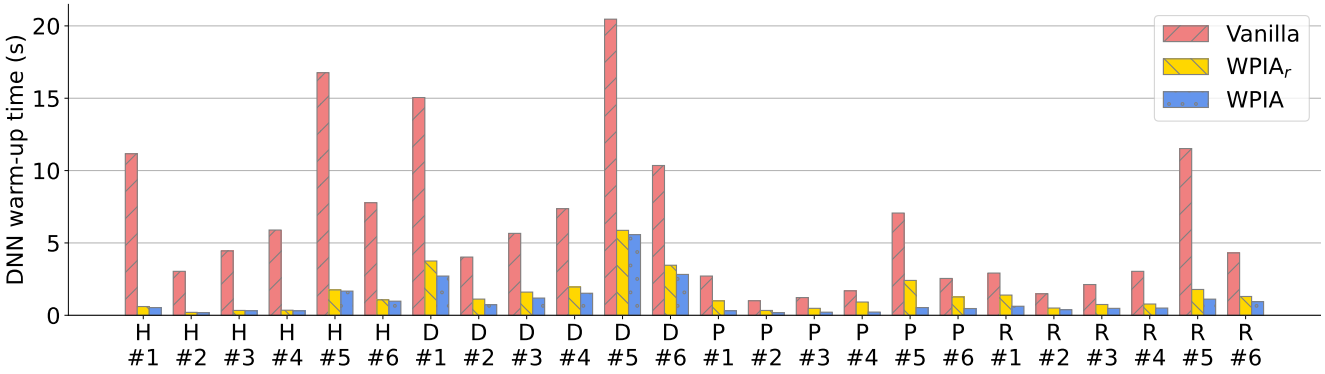


Fig. 7: The warm-up time. The X-axis labels are combinations of device IDs (H, D, P, R) and model IDs (#1 ~ #6).

Table 3: The DNN models used in evaluation.

ID	Name	# of operators	Size
#1	GoogLeNet	96	26.0 MB
#2	MobileNetV1	59	16.5 MB
#3	MobileNetV2	103	13.7 MB
#4	SqueezeNet	146	5.0 MB
#5	DenseNet	372	27.4 MB
#6	EfficientNetV2	404	27.0 MB

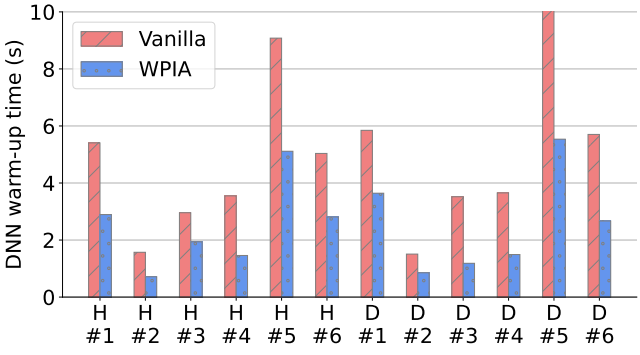


Fig. 8: The warm-up time on dedicated GPUs

and devices. For the vanilla approach, the warm-up time is equal to the first DNN inference latency. For WPIA$_r$ and WPIA, the warm-up time includes the time needed to download the WebGL binaries and load the binaries into Web browsers, as well as the first DNN inference latency. We repeat each experiment five times and report the average warm-up time. Figure 7 shows our experiment results about the DNN warm-up time of vanilla, WPIA$_r$, and WPIA.

We can see from Figure 7 that WPIA achieves surprising performance in accelerating DNN warm-up compared with vanilla. WPIA saves an average of 5.4 seconds in DNN warm-up and 15.1 seconds on maximum (DenseNet on the Hasee laptop). Regarding the ratio, WPIA reduces at least 72.7% of the DNN warm-up time in vanilla, with 84.1% on average and 95.3% on maximum (GoogLeNet on the Hasee laptop). In other words, WPIA can accelerate DNN warm-up from 3.7× to 21.1×. These results show that WPIA can optimize the DNN warm-up time remarkably.

Comparing the results among the devices, we can find that WPIA has different performances on different devices. The performance of WPIA on the Hasee laptop is much better than on other devices. The average speedup ratio on the Hasee laptop is 15.0×, whereas the average speedup ratio on other platforms is 6.0×. This phenomenon reflects that compiling WebGL programs is a severe performance bottleneck of DNN warm-up in Web browsers on some devices.

Dedicated GPUs. Some computers have dedicated GPUs to improve the performance of graphic rendering. While the default GPU choice for Web browsers is integrated GPUs, users can manually set the dedicated GPUs as their preferred GPUs in Web browsers. We conduct the same experiments on the two laptops using dedicated GPUs. The results are shown in Figure 8. From Figure 8, we can see that WPIA can still achieve large optimization on dedicated GPUs. WPIA reduces 2.1 seconds and 43.5% of the vanilla approach on average, and 4.8 seconds and 59.9% in maximum.

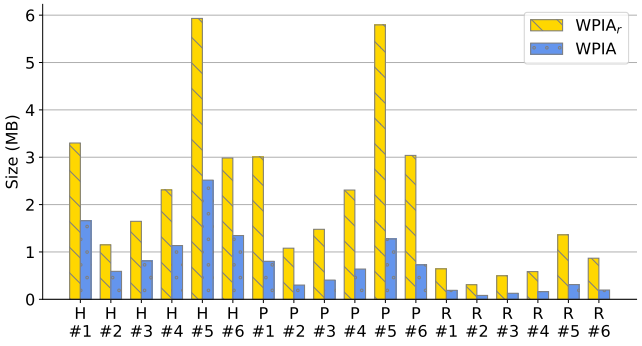
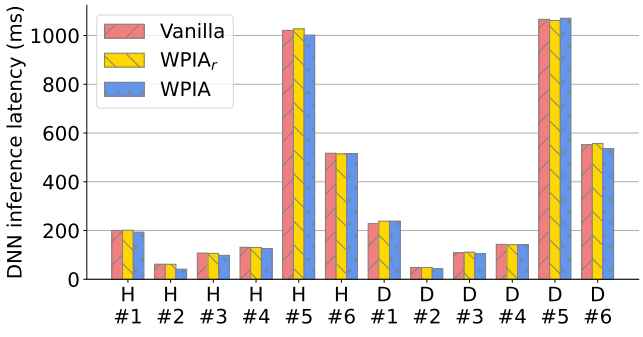
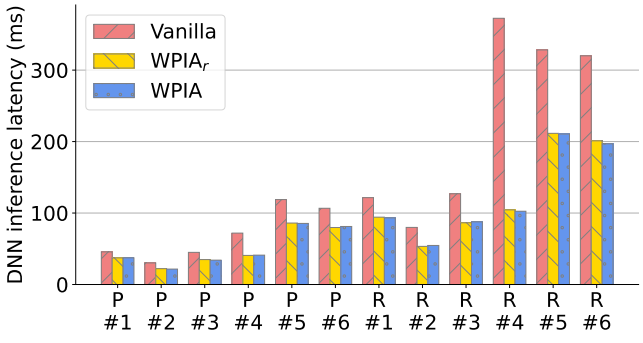


Fig. 9: The size of WebGL binaries



(a) Desktop browsers



(b) Mobile browsers

Fig. 10: The DNN inference latency (ms)

Takeaways: *WPIA has remarkable performance in optimizing the DNN warm-up time in Web browsers. On average, WPIA reduces 5.4 seconds or 84.1% of the DNN warm-up time of the vanilla approach.*

5.3 RQ2: How Does the WebGL Program Merging Contribute to WPIA?

Program merging shrinks the total size of the binaries of the WebGL programs, which can reduce the time fetching the precompiled WebGL programs. However, program merging also increases the size of a single WebGL program and adds additional code to the programs in execution. These factors can have positive and negative effects on the performance of WPIA.

Table 4: Comparison of WebGL programs of models before and after program merging

ID	# of programs		Source code size (MB)		Total # of functions	
	WPIA$_r$	WPIA	WPIA$_r$	WPIA	WPIA$_r$	WPIA
#1	198	8	1.32	0.69	6263	3167
#2	74	8	0.48	0.26	2242	1200
#3	107	7	0.7	0.38	3363	1785
#4	153	9	0.88	0.42	4141	1843
#5	396	9	2.26	0.83	10722	3812
#6	206	8	1.25	0.6	5953	2776

To answer RQ2, we compare the performance and overhead of WPIA and WPIA$_r$. First, we study the source code of WebGL programs of WPIA$_r$ and WPIA. Table 4 shows the statistics. From Table 4, we can see that program merging dramatically reduces the source code of WebGL programs. WPIA's merging effectively reduces the number of programs used in DNN warm-up from hundreds to less than ten. If we compare the total length of the source code, WPIA reduces around 50% of the total length after program merging, which will finally result in the reduced size of WebGL program binaries. The total number of functions is also greatly reduced by WPIA. WPIA reduces around half of the number of functions.

Program merging effectively reduces the total size of the WebGL binaries. We measure the total size of WebGL program binaries of WPIA$_r$ and WPIA, and Figure 11 displays the measurement results. As shown in Figure 9, the compression ratio of WPIA ranges from 22.0% to 51.4%, and 36.9% on average. In other words, program merging can reduce nearly half of the WebGL binaries in the worst case, effectively reducing the time that WPIA spends fetching and loading the binaries.

Comparing the effectiveness of program merging between different devices, we can find that the type of devices influences the effectiveness. On mobile phones, program merging is more effective than on laptops. The average compression ratio is 48.9% on laptops, while the average compression ratio on mobile devices is 25.9% after program merging. This phenomenon is probably due to the implementation difference of the WebGL program compilers on different GPUs.

Figure 7 shows the difference in DNN warm-up time between WPIA and WPIA$_r$. We can see that WPIA further improves the DNN warm-up time than WPIA$_r$. Compared with WPIA$_r$, WPIA further reduces 32.9% of the DNN warm-up time on average, and in the best case, it reduces 77.9% of the time, which implies that program merging can further reduce about one-third of the DNN warm-up time of WPIA$_r$.

Figure 10 compares the latency of WPIA and WPIA$_r$. We can see no noticeable difference in inference latency between the two approaches. This result indicates that program merging in WPIA does not introduce any noticeable overhead in

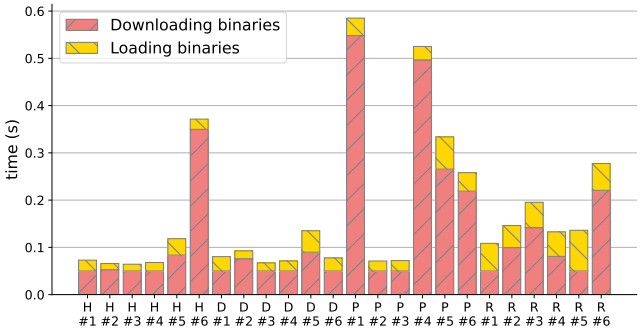


Fig. 11: The overhead of WebGL binaries in warm-up

DNN inference compared to WPIA$_r$.

***Takeaways**: Program merging in WPIA effectively reduces the total size of the WebGL source code and binaries, decreases the DNN warm-up time, and does not introduce any noticeable overhead in later DNN inference.*

5.4 RQ3: What Is the Overhead of WPIA?

Recalling the design of WPIA, we can find that WPIA introduces additional overhead in the following two aspects.

1. WPIA needs to download the binaries of the compiled programs, which is an additional step compared with vanilla frameworks.

2. WebGL Program merging increases the size of a single WebGL program binary, which might bring overhead to the execution of WebGL programs.

3. Precompiling WebGL programs may cause overhead for the server.

We analyze the time taken in downloading and loading binaries in WPIA in our experiments. Figure 11 shows the analysis results. From Figure 11, we can see that the time that WPIA takes in downloading and loading binaries is very little. The average time of downloading and loading is 0.17 seconds, which is trivial compared with the warm-up time of DNN models. Downloading binaries takes most proportion of the overhead. The average time of loading binaries is 0.04 seconds, which indicates that loading binaries into Web browsers incurs almost no overhead.

To show the overhead of WPIA's merging to the WebGL program execution, we measure execution time of WebGL programs during DNN inference. For each WebGL program execution in DNN inference, we repeat the execution 200 times, and measure the average execution time. We leverage the *EXT_disjoint_timer_query_webgl2* extension [39] to measure the execution time of WebGL programs on GPU devices. We carry out the experiment on the Dell laptop. The results show that the program merging increases the execution time of WebGL programs by 5.2μs, or by 21.5%, in the median case. We also sum up all time in executing WebGL programs for a single DNN inference, and compare the total execution time before and after program merging. The results show that program merging increase the total execution time 6.1ms, or by 17.5%, on maximum. These results show that program merging does not incur much overhead to WebGL program execution.

To show the overall influence of program merging, we also measure the average latency of DNN inference of WPIA. We repeat the DNN inference 100 times after the DNN warm-up with the vanilla approach and WPIA. The results are shown in Figure 10. From Figure 10, we can see no obvious latency increase between the vanilla approach and WPIA. Moreover, on mobile devices, WPIA reduces the latency of DNN inference compared with vanilla. The average latency drop is 33.2% for mobile devices. The optimization stems from the record-and-replay mechanism in WPIA, which avoids generating any source code of WebGL programs in DNN inference.

To show the overhead of WPIA's precompiling to the server, we measure the time that the server takes to compile the merged WebGL programs of different DNN models for different user devices. We repeat each measurement three times and report the average results. The results are shown in Table 5. We can see that precompiling WebGL programs into binaries takes only very little time. Among these experiments, the shortest time is less than 1 second (MobileNetV1 for Redmi), and the longest is less than 2 minutes (DenseNet for Dell). The average compiling time is 17.6 seconds. The results show that WPIA can accomplish WebGL program precompiling fairly fast, which implies that the precompiling of WPIA will not incur much overhead to the servers in the offline phase.

***Takeaways**: The overhead of downloading WebGL binaries is little, and the latency of DNN inference does not increase in WPIA. The precompiling time in the offline phase is almost negligible. Therefore, WPIA will not incur much overhead in practice.*

6 Discussion

This section will discuss some possible issues that may arise when deploying WPIA in real-world scenarios.

- *Security concerns.*

There might be security concerns if a Web browser allows an arbitrary binary from the Internet to run directly on the GPU in a user device. However, in practice, using a WebGL program loaded from a precompiled binary poses no more security threats than using a program compiled from source code. Possible threats from executing an arbitrary WebGL binary, such as out-of-range memory access and denial of service, have long been a security concern. Currently, OpenGL extension *GL_ARB_robustness* [40] defends against these possible threats from WebGL programs, and the defense also applies to the programs loaded from binaries. Yao et al. [41] leverages GPU virtualization to isolate the execution of WebGL programs, which can also defend against malicious precompiled WebGL programs.

Table 5: Time of WebGL Program Precompiling (s)

Model ID	Hasee laptop	Dell laptop	Pixel phone	Redmi phone
#1	34.5	31.9	4.0	9.3
#2	5.5	4.6	0.8	0.9
#3	11.0	9.2	2.5	4.0
#4	12.3	10.3	1.4	3.1
#5	88.6	101.1	12.0	35.3
#6	15.7	17.0	2.7	5.0

We believe that Web browsers should impose strict security policies on using precompiled WebGL programs. For example, Web apps that need this API should be hosted on HTTPS, which can protect from Man-in-the-Middle attacks that try to modify the binaries during network transfer. The browser should impose integrity checking to ensure the binaries have not been tampered with. With strict security policies, loading a precompiled WebGL program binary from the Internet will not create new security threats to users.

- *Cost of precompiling*.

Because the format of a WebGL program depends on the software stack in Web browsers and the vendor of the GPU device, a WebGL program may have different binaries on different GPU devices. Generally speaking, compiling a WebGL program for a specific GPU requires the help of real GPU devices. If a developer wants to load the precompiled binaries on various user devices, intuitively, it requires the developer to run WPIA's offline phase on various real GPU devices, which may bring cost in deploying WPIA.

Multiple feasible workaround approaches exist to compile WebGL programs without specific GPU devices. Tools such as Mali offline compiler [42] from AMD can compile a WebGL program without the specific GPUs. By default, Chromium-based browsers and Firefox on Windows compile WebGL programs into Direct3D binary format [24] supported by Windows, so these binaries are portable to other Windows devices with different GPU devices. Furthermore, if a user's Web browser supports extracting the binary of a WebGL program, we can acquire the binaries of compiled WebGL programs from the user's Web browser. Then, the next time, these WebGL binaries can be used to accelerate the DNN warm-up for other users with the same GPU devices.

7 Related Work

In this section, we will briefly introduce the related work of this paper.

7.1 Deep Learning in Web Apps

With the recent development of deep learning, more and more Web apps are starting to leverage DNN inference to provide intelligent service to users. Quan et al. [43] investigate the faults of JavaScript-based deep learning frameworks. Our prior work [15] measure the performance of DL frameworks in DNN inference, with various inference backends and on different DNN models. They also identify that DNN inference on the WebGL backend has a long warm-up time, but they do not further analyze the reason for this phenomenon. Huang et al. [44] propose DeepAdapter, which adopts a context-aware DNN pruning scheme to perform DNN inference collaboratively. The WEngine tool proposed by us [45] can improve the DNN inference throughput for Web apps by scheduling DNN inference tasks on WebGL and WebAssembly backends. Huang et al. [46] design AoDNN, which is a delicate framework for offloading DNN inference tasks from Web browsers to edge servers. Although many research efforts have been invested in optimizing DNN inference performance, to the best of our knowledge, there is no work that specifically targets optimizing the DNN warm-up.

7.2 WebGL Optimization

WebGL is a powerful API that allows web apps to utilize the GPU resources in user devices. Prior work on WebGL has focused on discovering and fixing security issues. For example, it has been shown by Tahir et al. [47] that WebGL APIs can be exploited to do cryptojacking in Web browsers. Yao et al. [41] propose Sugar, which isolates the WebGL context from different Web apps with GPU virtualization. We believe their approach can also be used to protect malicious behavior from WebGL programs loaded from binaries.

Wu et al. [48] propose UNIGL which can rewrite WebGL programs to defend against Website fingerprinting with WebGL. Some OpenGL shader compilers apply different levels of source-code optimization on the shaders [49], including dead-code elimination, and loop unroll. Approach proposed by Han and Abdelrahman [50] can reduce branch divergence in GPU code, as well as moving out common code from conditional expressions. To the best of our knowledge, we conduct the first work that tries to reduce the total size of multiple compiled WebGL programs.

8 Conclusion

In this paper, we study the long warm-up time of GPU acceleration of DNN inference in Web browsers. We analyzed the reason behind the long warm-up time through a measurement study and revealed that compiling WebGL programs takes most of the warm-up time. Inspired by this finding, we proposed WPIA, an approach that suggests precompiling WebGL programs at the server side to avoid compiling them in Web browsers. WPIA tackles challenges of precompiling by merging WebGL programs and using a record-and-replay technique. We implemented a prototype of WPIA and evaluated its performance and overhead. Experiment results show that WPIA can accelerate the DNN warm-up time to an order

of magnitude with negligible additional overhead.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 62102009) and Beijing Outstanding Young Scientist Program (Grant No. BJJWZYJH01201910001004).

Competing interests The authors declare that they have no competing interests or financial conflicts to disclose.

References

1. Google . Background Features in Google Meet, Powered by Web ML. https://ai.googleblog.com/2020/10/background-features-in-google-meet.html?m=1, 2020
2. Amazon . BackgroundFilter_Video_Processor. https://aws.github.io/amazon-chime-sdk-js/modules/backgroundfilter_video_processor.html, 2023
3. Google . Google Meet. https://meet.google.com/?pli=1, 2023
4. Amazon . Video Conferencing & Online Meetings - Amazon Chime. https://aws.amazon.com/chime/, 2023
5. Snapchat . Snapchat, now on the Web. https://web.snapchat.com/, 2023
6. Adobe . Adobe Creative Cloud. https://creativecloud.adobe.com/cc/photoshop, 2023
7. Adobe . How Adobe used Web ML with TensorFlow.js to enhance Photoshop for web. https://blog.tensorflow.org/2023/03/how-adobe-used-web-ml-with-tensorflowjs-to-enhance-photoshop-for-web.html, 2023
8. Compilation M L. Web Stable Diffusion. https://github.com/mlc-ai/web-stable-diffusion, 2023
9. Compilation M L. Web LLM. https://github.com/mlc-ai/web-llm, 2023
10. Group K. WebGL - OpenGL ES 2.0 for the Web. https://www.khronos.org/webgl/, 2021
11. Consortium W W W. Webgpu. https://www.w3.org/TR/webgpu/, 2023
12. Smilkov D, Thorat N, Assogba Y, Yuan A, Kreeger N, Yu P, Zhang K, Cai S, Nielsen E, Soergel D, others . Tensorflow.js: Machine learning for the web and beyond. arXiv preprint arXiv:1901.05350, 2019
13. Microsoft . ONNX.js: run ONNX models using JavaScript. https://github.com/microsoft/onnxjs, 2021
14. Hidaka M, Kikura Y, Ushiku Y, Harada T. WebDNN: Fastest DNN Execution Framework on Web Browser. In: Liu Q, Lienhart R, Wang H, Chen S K, Boll S, Chen Y P, Friedland G, Li J, Yan S, eds, Proceedings of the 2017 ACM on Multimedia Conference, MM 2017, Mountain View, CA, USA, October 23-27, 2017. 2017, 1213–1216
15. Ma Y, Xiang D, Zheng S, Tian D, Liu X. Moving Deep Learning into Web Browser: How Far Can We Go? In: The World Wide Web Conference, WWW '19. 2019, 1234–1244
16. TensorFlow . Platform and environment | TensorFlow.js. https://www.tensorflow.org/js/guide/platform_environment, 2021
17. 2023 Digital Experience Benchmark | Contentsquare. https://contentsquare.com/insights/digital-experience-benchmark/, 2021
18. Akamai Performance Matters Key Consumer Insights Ebook. https://www.scribd.com/document/300122029/Akamai-Performance-Matters-Key-Consumer-Insights-eBook, 2023
19. Website Load Time Statistics (2024): Average Page Load Time & Bounce Rate. https://www.tooltester.com/en/blog/website-loading-time-statistics/, 2024
20. Munk O D, Scoccia G L, Malavolta I. The state of the art in measurement-based experiments on the mobile web. Inf. Softw. Technol., 2022, 149: 106944
21. Google . The Chromium Projects. https://www.chromium.org/, 2022
22. Mozilla . <canvas>: The Graphics Canvas element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas, 2023
23. Gao P, Xu Y, Song F, Chen T. Model-based automated testing of javascript web applications via longer test sequences. Frontiers Comput. Sci., 2022, 16(3): 163204
24. Google . Angle. https://github.com/google/angle, 2022
25. Kim Y, Kim J, Chae D, Kim D, Kim J. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In: Proceedings of the Fourteenth EuroSys Conference 2019. 2019, 1–15
26. Jia F, Zhang D, Cao T, Jiang S, Liu Y, Ren J, Zhang Y. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices. In: Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, MobiSys '22. 2022, 209–221
27. Mil webdnn. https://mil-tokyo.github.io/webdnn/, 2023
28. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S E, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. 2015, 1–9
29. Howard A G, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreetto M, Adam H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017
30. Sandler M, Howard A, Zhu M, Zhmoginov A, Chen L C. Mobilenetv2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2018, 4510–4520
31. Iandola F N, Han S, Moskewicz M W, Ashraf K, Dally W J, Keutzer K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. arXiv preprint arXiv:1602.07360, 2016
32. Iandola F N, Moskewicz M W, Karayev S, Girshick R B, Darrell T, Keutzer K. DenseNet: Implementing Efficient ConvNet Descriptor Pyramids. CoRR, 2014, abs/1404.1869
33. Tan M, Le Q V. EfficientNetV2: Smaller Models and Faster Training. In: Meila M, Zhang T, eds, Proceedings of the 38th International Con-

ference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event. 2021, 10096–10106

34. Zhang L L, Han S, Wei J, Zheng N, Cao T, Liu Y. nn-METER: Towards Accurate Latency Prediction of DNN Inference on Diverse Edge Devices. GetMobile Mob. Comput. Commun., 2021, 25(4): 19–23
35. Find Pre-trained Models. https://www.kaggle.com/models?tfhub-redirect=true&framework=tfJs, 2023
36. Büch L, Andrzejak A. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: Wang X, Lo D, Shihab E, eds, 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019. 2019, 95–104
37. Farmahinifarahani F, Saini V, Yang D, Sajnani H, Lopes C V. On precision of code clone detection tools. In: Wang X, Lo D, Shihab E, eds, 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019. 2019, 84–94
38. Ferreira F, Valente M T. Detecting code smells in React-based Web apps. Inf. Softw. Technol., 2023, 155: 107111
39. WebGL EXT_disjoint_timer_query_webgl2 Extension Specification. https://registry.khronos.org/webgl/extensions/EXT_disjoint_timer_query_webgl2/, 2023
40. Group K. GL_ARB_robustness. https://registry.khronos.org/OpenGL/extensions/ARB/ARB_robustness.txt, 2022
41. Yao Z, Ma Z, Liu Y, Amiri Sani A, Chandramowlishwaran A. Sugar: Secure GPU acceleration in web browsers. ACM SIGPLAN Notices, 2018, 53(2): 519–534
42. ARM . Mali Offline Compiler. https://developer.arm.com/Tools\%20and\%20Software/Mali\%20Offline\%20Compiler, 2022
43. Quan L, Guo Q, Xie X, Chen S, Li X, Liu Y. Towards Understanding the Faults of JavaScript-Based Deep Learning Systems. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. 2022, 105:1–105:13
44. Huang Y, Qiao X, Tang J, Ren P, Liu L, Pu C, Chen J. DeepAdapter: A Collaborative Deep Learning Framework for the Mobile Web Using Context-Aware Network Pruning. In: IEEE INFOCOM 2020. 2020, 834–843. ISSN: 2641-9874
45. Tian D, Shen H, Ma Y. Parallelizing DNN inference in mobile web browsers on heterogeneous hardware. In: MobiSys '22. 2022, 519–520
46. Huang Y, Qiao X, Dustdar S, Li Y. AoDNN: An Auto-Offloading Approach to Optimize Deep Inference for Fostering Mobile Web. In: IEEE INFOCOM 2022. 2022, 2198–2207
47. Tahir R, Durrani S, Ahmed F, Saeed H, Zaffar F, Ilyas M S. The Browsers Strike Back: Countering Cryptojacking and Parasitic Miners on the Web. In: 2019 IEEE Conference on Computer Communications, INFOCOM 2019. 2019, 703–711
48. Wu S, Li S, Cao Y, Wang N. Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting. In: Heninger N, Traynor P, eds, 28th USENIX Security Symposium, USENIX Security 2019. 2019, 1645–1660
49. Crawford L, O'Boyle M F P. A Cross-platform Evaluation of Graphics Shader Compiler Optimization. In: IEEE ISPASS 2018. 2018, 219–228
50. Han T D, Abdelrahman T S. Reducing branch divergence in GPU programs. In: Proceedings of 4th Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2011, Newport Beach, CA, USA, March 5, 2011. 2011, 3