**基于WebAssembly和WebGL浏览器播放H.265编码视频的系统研究**

缑通旺

## **摘 要：**针对大部分浏览器因为H.265编码专利问题而不支持对H.265编码类型视频的播放，设计基于WebAssembly和WebGL的浏览器对H.265编码视频解码、渲染的系统。本文首先介绍了该系统的整体架构以及功能模块架构，并通过研究利用FFmpeg开源音视频解码库，提出一种基于迁移WebAssembly编码方案提高浏览器音视频解码速度以及WebGL图像渲染引擎渲染解码后的视频帧数据流的方法。该方法实现了在浏览器环境中对FFmpeg 的C语言解码库的调用执行，以及提高了解码速度、渲染性能，优化了视频播放的性能体验。最后实验也表明该系统的视频播放方案可行且优化效果良好，同时对CPU密集型任务处理的代码库迁移到浏览器中使用提供了理论验证和实践基础。

**关键词：** WebAssembly； WebGL；FFmpeg； H.265； 浏览器；

**Playing H.265 Encoded Video By Browser Based On WebAssembly And WebGL**

Gou Tongwang

**Abstract:** For most browsers do not support the playback of h.265 encoded video because of the h.265 encoding patent, a browser playback system based on WebAssembly and WebGL is designed. Firstly, this paper introduces the overall architecture and functional module architecture of the system, and puts forward a method to improve the browser decoding speed based on WebAssembly coding scheme and render the decoded video frame data stream by using FFmpeg open source decoding library. This method realizes the execution of FFmpeg C language decoding library by the browser, improves the decoding speed and rendering performance, and optimizes the performance experience of video playback. The final experiment shows that the video playback scheme of the system is feasible and the optimization effect is good. At the same time, it provides theoretical verification and practical basis for the migration of CPU intensive task processing library to browser.

**Keywords:** WebAssembly； WebGL；FFmpeg； H.265； Browser；

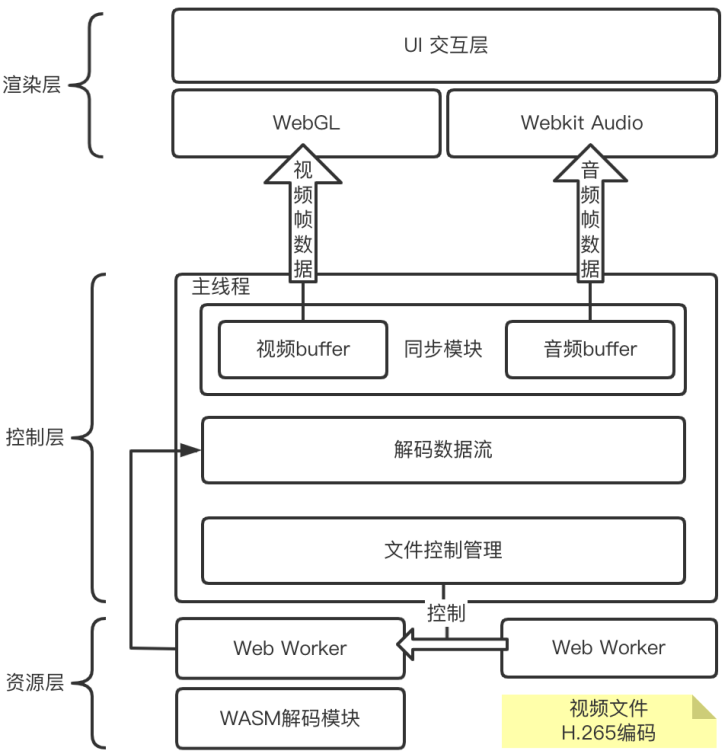
随着音视频深度渗透、全面融入人们生活，音视频已经成为网民表达自身情感、生活和想法的重要工具。因为人们对视频的清晰度体验要求越来越高，随着1080p、4k、8k等高分辨率清晰度的出现给视听体验带来了更加清晰的图像画质、更加丰富的细节，但是同时带来了体积不断膨胀的视频流文件，给视频流在传输、存储带来了巨大的压力。而在音视频的技术领域中，与现在主流的视频压缩H.264/AVC相比，H.265编解码的高压缩比特性可以实现节省一半左右的存储空间还得到相同的视频质量，从而显著的降低了高分辨率视频的存储压力和传输压力[1]。尽管其编解码的优异特性，但是因为其专利问题，市场上大部分浏览器并不支持H.265编码。

在音视频播放应用的研究中，B/S架构相对于C/S架构有很多先天缺陷，比如B/S架构只能执行JavaScript脚本语言，而这种解释型脚本语言并不适合处理CPU密集型任务，也给音视频在浏览器中处理带来挑战。但随着Web领域的高速发展，在复杂的业务需求中需要浏览器承担一部分CPU密集型任务的处理，所以很多科学家和工程师为了让代码在浏览器中跑的更快而不断探索。2011年 Google 创造了 Native Client（NACI）到从2013年Mozilla提出ASM.js可以很大程度的优化和提高执行速度；再到Mozilla、谷歌、微软以及苹果联合定义了WebAssembly，致力于为各个语言定义一种二进制形式的编译目标格式，并设计一种可以与Web平台集成与执行的方案[2]。

传统的Web视频播放是通过Canvas、Audio进行渲染播放[3]，首先这种方案是基于浏览器内置解码所以不支持H.265编码，而且这种方案具有延时高、帧率不稳定、渲染性能差、编码格式支持有限等问题，很难满足实际生产中对视频播放的灵活要求。针对这个问题，本文通过迁移WebAssembly编码[4-5]对FFmpeg解码库编译[6-7]，并结合WebGL[8]技术实现对H.265码流视频渲染的播放系统，并优化图像渲染性能、解码性能。最终根据本文的方法可以实现浏览器对H.265编码视频的播放功能，且播放视频具有低延时性，解码、渲染性能高，可控性强等优点。

## 1系统整体架构

针对浏览器中对H.265编码视频解码、渲染播放性能的提高和功能实现，结合WebAssembly编码和WebGL渲染技术实现的视频播放系统架构如图1所示：

图1 系统架构结构

资源层：主要是对视频文件、WASM解码模块文件的抽象，通过Web Worker线程加载不同的资源模块进行数据读取、处理、输出的功能。

控制层：控制层主要分为文件控制管理模块、解码数据流模块、同步控制模块，其中文件控制管理模块主要对资源层文件处理协调的管理。解码数据流模块主要是对资源层音视频文件解码处理后对帧数据流的Buffer存储。同步模块主要是控制对视频Buffer和音频Buffer的数据读取消费同步以及和解码速度保持协调。

渲染层：主要是分别通过WebGL和Audio模块读取不同的Buffer数据流进行消费，最后提供给UI交互层。

以上是系统的整体架构，架构设计总体采用模块化设计，各个模块之间分工明确、协调统一，共同实现系统的功能目标。最终一个音视频文件在浏览器中实现渲染播放的流程可以分为如图2所示的具体过程。

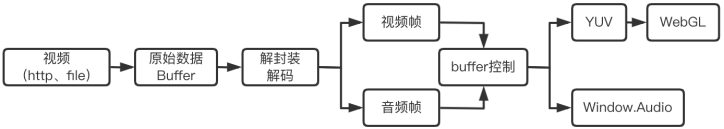


图2 音视频解码播放流程

## 2系统关键技术

**2.1 基于FFmpeg 自定义解码**

FFmpeg是一个开源免费的跨平台音视频分离、转换、解码于一体的基础库，同时包含了对流媒体的格式转换，媒体协议的转变、音视频的码率控制，采样率的改变以及色彩格式的修改等。FFmpeg支持MPEG、Divx、MPEG-4、FLV等40多种编码方式，以及AVI、OGG、Matroska、ASF等90多种解码方式。FFmpeg的开发基于Linux操作系统，并且可以在大多数操作系统中编译和使用，包括Windows平台、MacOS平台甚至是安卓平台等[7]。因为其开源性、良好的跨平台性以及可移植等特点，得到了广泛应用，MPlayer、VLC以及国内QQ影音等播放器都对FFmpeg开源库进行了参考和二次开发。

本小节主要内容是基于FFmpeg的解码模块做二次开发从而实现对H.265编码类型音视频文件的解码功能以及对音视频帧同步渲染控制的目的。

2.1.1 双层数据Buffer

系统在读取音视频文件和解码处理后分别定义了数据缓冲Buffer层，简称为文件数据Buffer和解码数据Buffer。其中文件数据Buffer主要是对不同协议的音视频文件进行数据读取缓存。当浏览器中视频播放页面初始化的时候，会先请求到音视频文件的元信息，包含视频文件的大小、编码元信息等，用于动态定义Buffer的大小。当音视频文件的数据读取达到一定阈值时解码模块开始读取数据进行解码，并存储到解码数据Buffer中，其中解码数据Buffer主要分为两部分：图像数据和音频数据。而WebGL和Audio分别从解码后的数据Buffer中读取YUV图像数据和PCM数据进行消费，渲染图像以及播放音频。

通过两层数据缓冲Buffer，实现当音视频开始播放时不会因为数据未解码或数据不足导致系统崩溃。同时控制音视频数据读取不够时停止解码，进入音视频文件数据缓存状态。当音视频数据缓冲足够后，开启数据帧的解码播放，最终才能保证音视频播放的稳定性。

2.1.2 自定义同步控制模块

同步控制模块主要实现音画同步的渲染播放。通过上一小节，系统可以从解码数据的Buffer中获取到音频帧数据和视频帧数据，但需要同步控制模块保证音频帧和视频帧都按照正确的逻辑时间点进行渲染播放，并对于音频帧和视频帧的时间戳进行校验，修正错误的逻辑时间戳。

系统选取音频帧为基准，通过Web Audio获取到音频帧每一帧应该出现在视频中的时间戳，在解码视频帧的过程中通过公式计算得到当前视频帧的时间戳。通过比较当前视频帧和音频帧的时间戳，如果视频帧的时间已经落后则立即渲染；如果视频帧的时间相比较过早，则对视频图像的渲染进行延迟。通过这种方式实现音频帧和视频帧数据的同步消费。

式中pts是主要用于度量解码后的视频帧什么时候被显示出来，av\_q2d(time\_base)是每个时间刻度代表的真实时间间隔。

同步控制模块还需要控制解码速度和视频播放之间的关系。具体表现为不能出现视频暂停后还会继续一直解码；不能出现视频刚开始播放才开始解码；不能出现视频播放的速度快于解码速度。

通过同步控制模块和数据双层缓冲Buffer可以实现对音视频文件读取、解码、消费的三个步骤的协调执行。

**2.2 WASM编码技术**

WASM是一个可移植、体积小、加载快、兼容型强，且拥有全新编码格式的二进制字节码[9]，它可以在现代网络浏览器中直接运行，是一种低级的类汇编语言，具有紧凑的字节码格式，接近原生的性能运行，可以称为“浏览器中的汇编语言”。目前WASM仍处于发展阶段，但针对WASM的研究和应用一直处于广泛关注的状态。WASM适合用于大量计算的场景，例如以下场景：处理音视频，Flv.js用WASM重写后性能有很大提升；Tensorflow.js一种在浏览器中训练和推理模型的技术也利用了WASM来加快模型训练、推理、可视化等等场景。WASM目前被大多数浏览器厂商、多种编程语言支持，并且广泛应用于各种高性能容器场景，嵌入式系统以及边缘计算，同时尤其是给在Web技术架构下处理CPU密集型任务打开了一扇大门。

2.2.1 WASM的堆栈虚拟机和编码模型

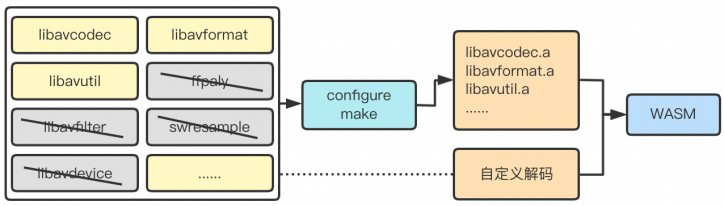
Wasm之所以会有如此高的加载和运行效率，离不开Web浏览器在其底层对WASM代码的独特解析和执行过程。WASM代码在底层编译器内部的解析和处理流程可以抽象为一种堆栈式虚拟机[10]。除此之外，WASM还定义了一套虚拟指令集，通过该虚拟指令集可以将WASM二进制编码通过编译器后端转译为平台相关的浏览器汇编代码再进行执行。在这个过程中，相比较JavaScript代码的解释执行，WASM代码的执行只需要最后一步编译为汇编代码，所以执行速度会大幅度提升。

WASM采用了基于小端模式的编码算法进行可变长编码，可以在一定程度上保证模块体积大小处于最优的状态。其中对于不同的数据采用不同的编码方案，例如用于整数的LEB-128编码、用于浮点数的IEEE-754编码以及用于字符串的UTF-8编码。

2.2.2 利用WASM编译FFmpeg解码模块

系统将视频解码这个CPU密集型操作的处理编译为WASM模块，在浏览器中使用Web Worker加载并调用来提高浏览器对这类操作的性能，从而为H.265编码的音视频播放提供了保障。对于FFmpeg采用定制化编译的思路，FFmpeg框架的基本组成包含AVFormat、AVCodec、AVFilter、AVDevice以及AVUtil等模块，针对系统功能目标裁剪选取系统中所需要的avcodec、avformat、avutil等模块，这样做的好处可以减少最终生成的WASM模块的体积，进一步优化模块的加载速度和执行速度。

通过阅读FFmpeg源码，FFmpeg提供了对自身模块的定制化编译选项。通过类似--disable-avfilter，--disable--ffprobe可以实现编译裁剪avfilter、ffprobe模块的目的。同时需要指定--cc=”emcc” 等一系列编译参数来确定编译工具、类型等，最终定制化编译FFmpeg和自定义解码模块的流程如图3所示，

图3 定制化编译流程模型

在configure make配置编译参数的模块，系统需要实现FFmpeg二次开发模块和浏览器环境中的部分函数可以相互调用，从而保障浏览器可以完整调用自定义解码功能。如表1所示，通过EXPORTED\_FUNCTIONS关键字指定WASM模块中函数名列表都可以在浏览器环境中通过WASM模块的引用直接获取并使用。通过EXTRA\_EXPORTED\_RUNTIME\_METHODS指定的addFunction关键字，可以通过addFunction定义函数并在WASM模块中动态获取并执行。

表 1 JavaScript和WASM实现相互调用的方式

|  |  |
| --- | --- |
| 功能 | 具体参数 |
| JavaScript调用WASM | EXPORTED\_FUNCTIONS=  [“methodName”] |
| WASM  调用JavaScript | EXTRA\_EXPORTED\_RUNTIME  \_METHODS=[“addFunction”] |

本小节通过的FFmpeg源码阅读采用定制化编译的思路并设计编译流程方案，并可以实现在浏览器中调用二次开发好的功能函数，以及可以在WASM模块中动态获取并处理浏览器中自定义的方法的功能。至此，作为H.265编码的音视频播放系统的基础核心已经全部实现。

**2.3 WebGL渲染技术**

WebGL是一个JavaScript实现的API，可以在任何兼容Web浏览器中渲染高性能的交互式3D和2D图形。该API可以在HTML5的元素中使用，同时利用了用户设备提供的硬件图像加速，即GPU加速。WebGL中的核心概念是顶点着色器和片元着色器。将顶点着色器和片元着色器连接起来的方法叫做着色程序。顶点着色器的作用是计算顶点位置，即提供顶点在裁剪空间中的坐标值。片元着色器的作用是计算图元的颜色值。利用WebGL来渲染图像的目的是通过GPU加速来实现浏览器渲染性能的提升。

2.3.1 RGB和YUV

浏览器渲染图像涉及到颜色空间的概念。目前最常见的颜色空间主要是RGB和YUV两种[11]。

RGB通常是指三原色即红绿蓝，通过这三种基本色彩可以组成其他的颜色。视频其实是由数量极大的帧画面一帧帧播放组成，当播放帧率达到一定大小，人的肉眼就无法感觉帧画面的切换。而RGB是通过对每一个画面的像素点分为R通道、G通道、B通道，通过这三种通道的值渲染出图片。

YUV颜色空间中，Y指代明亮度、U和V分别指代色度，作用是描述影像色彩及饱和度，用于指定像素的颜色。人们肉眼对明亮度远大于对色彩的敏感程度，利用这种敏感度的差异，所以可以对U、V颜色空间提高压缩比例，减少压缩后的体积，同时不会让肉眼察觉到前后的明显差异。所以YUV相对于RGB的压缩比例要高，也就是说YUV格式比RGB格式存储空间更小、占用带宽更小、更适合存储和传输。

2.3.2 WebGL绘制YUV

对Y、U、V三色空间，分别初始化一个WebGL画笔，然后先去绘制Y颜色空间的帧数据，然后根据U颜色空间数据的大小和U颜色空间数据的偏移量，用U-GL画笔去绘制，最后根据U和V颜色空间数据的大小以及偏移量最后绘制V颜色。最终相当于每一个像素点都绘制了三遍，然后三种属性叠加形成最后的图像像素点的颜色。

## 3实验

实验工具：Mac 2GHz 四核 Intel Core i5， 图形卡：Intel lris Plus Graphics 1536MB

运行环境：谷歌Google浏览器 x86\_64 版、Node v16.3.0

**3.1 WebGL和Canvas的性能对比实验**

现代浏览器都为Canvas 2d和WebGL都提供硬件加速的支持。 本小节针对WebGL和Canvas 2D渲染粒子数量梯度进行对比实验，分别使用WebGL和Canvas 2D绘制1000、5000、10000数量级粒子查看浏览器绘制页面时的FPS（Frame Per Second）的波动情况。选取了实验过程中具有代表性的Canvas在1000粒子如图4、10000粒子如图5和WebGL在10000粒子如图6的渲染情况所示。



图4 Canvas 1000粒子渲染情况

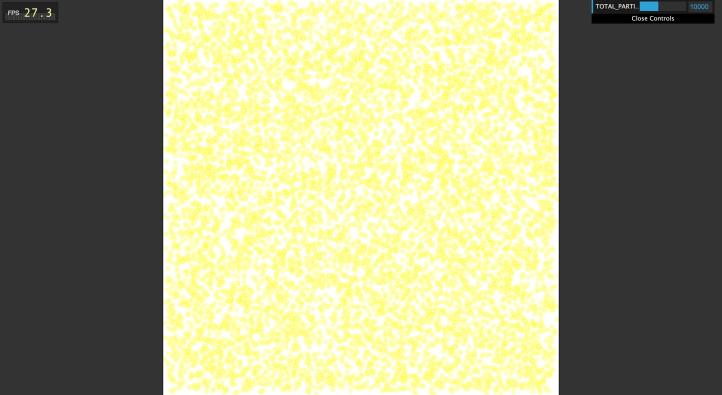


图5 Canvas 10000粒子渲染情况

****图6 WebGL 10000粒子渲染情况

从上图可以清楚的看到Canvas在粒子数量增多时其FPS数值在降低，同时图5看起来比图6的粒子数量密集的原因就是FPS值低导致的绘制重叠。最终实验结果汇总如下表所示：

表 2 Canvas和WebGL在粒子渲染的表现情况

|  |  |  |  |
| --- | --- | --- | --- |
| 数量 | 1000 | 5000 | 10000 |
| Canvas | 60.3fps | 53.5fps | 27fps |
| WebGL | 60.2fps | 60.1fps | 59.9fps |

通过以上实验内容可以证明WebGL相比较传统的Canvas渲染在粒子数量梯度上涨的同时，其FPS几乎没有什么波动，而Canvas在渲染粒子数量达到10000时，FPS已经失帧掉到了27fps左右，表现为渲染性能在高数量粒子情况下非常低，已经严重影响到用户的使用体验。本小节实验可以从侧面反应利用WebGL渲染帧图像画面的性能是远远好于Canvas。

**3.2 WebAssembly编码执行速度性能实验**

通过WASM将FFmpeg编译后的模块，通过Web Worker加载进浏览器后供JavaScript调用。构建WASM编码，首先要下载Emscripten工具链、CMake高级语言编译工具，本课题采用C所以使用GCC或者Clang编译器以及python，python主要是用来充当编译过程的一些脚本功能。

3.2.1 Hello World

利用搭建好的编译环境，先对简单的C语言打印Hello World程序进行编译，并运行在Node环境和浏览器环境下查看结果；

通过emcc命令也就是安装Emscrpten工具链增加的全局Command命令，执行emcc hello.c -o hello.js，当前目录下hello.c就会被编译，增加两个文件一个是hello.js，另一个是hello.wasm，其中hello.js是emcc帮助我们生成了一层胶水程序用来调用hello.wasm模块。通过Node，一种服务端的JavaScript执行框架来执行node hello.js，如图4.1所示通过C编写的函数代码编码为WASM后可以通过JavaScript执行调用并运行正确；同样如果4.2所示在浏览器环境下打开控制台，也同样可以运行成功。



图7 Node执行结果

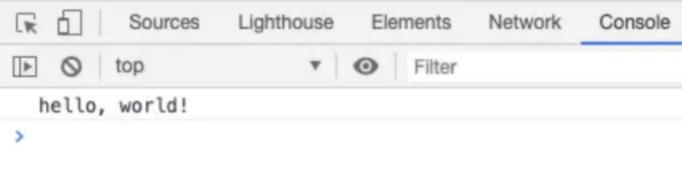


图8 浏览器执行结果

3.2.2斐波那契数列函数

针对上一小节的实验，针对JavaScript、C语言以及C语言编译为WASM后的斐波那契额数列函数进行梯度对比，查看针对这种超深递归的CPU密集型计算WASM编码能否带来性能上的提升与优化。

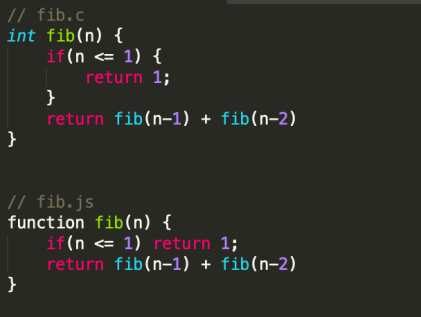


图9 斐波那契数列函数C语言和JavaScript版本

通过emcc命令将fib.c文件单独编译为WASM模块，并在JavaScript中引入调用。

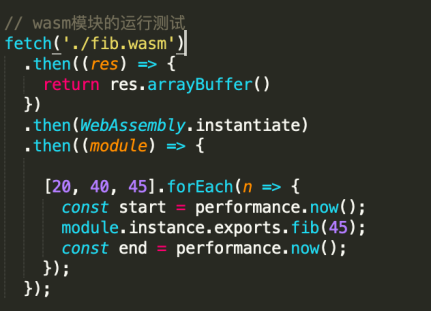


图10 JavaScript加载WASM函数

最终经过在Google浏览器、Mac 2GHz 四核Intel Core i5的同等环境下测试，得出如表4.1所示的相关数据，可以显著的观察到C以及C-WASM相比较JavaScript的执行时间几乎提高了45～47%的范围程度。

表 3 JavaScript、C、C-WASM运行结果对照

|  |  |  |  |
| --- | --- | --- | --- |
| 斐波那契函数 | JavaScript | C | C-WASM |
| 20 | 0.70ms | 0.00ms | 0.00ms |
| 40 | 1284.20ms | 682.443ms | 669.60ms |
| 45 | 14155.60ms | 7513.88ms | 7983.00ms |

根据本小节内容，可以总结以下两点：

（1）、C语言等编译型高级语言确实可以通过WASM编码为新型的独立二进制字节码，并可以在JavaScipt执行环境中正常使用。

（2）、WASM编码格式的程序相比较纯JavaScript代码执行上保留了编译型语言高效的执行效率。

尽管本小节的实验存在一定的误差因素，考虑的纬度还不够全面。但足以证明WASM编码可以对FFmpeg源代码进行编译处理后，在JavaScript中运行且可以拥有良好的执行效率，为本文系统实现以及后续工作打下了坚实的理论基础。

4.3 基于WASM和WebGL的视频播放系统测试

通过本文介绍的系统架构和技术方案搭建基于WASM和WebGL的浏览器H.265编码视频播放系统，在本地通过Node Http-Server工具启动服务，打开localhost:8080查看视频播放页面。

选取大小为25619346字节H.265编码进行测试，每一帧图像的大小计算公式为：

视频播放页面大小为宽852，高480，pix\_fmt是视频的像素格式为1.5，所以每一个帧的图像大小为613440字节。对于网络上的视频、大多数电视和电影，每秒 24 帧是行业中的标准。经测试总体上视频播放的稳定性和流畅度都符合正常视频的播放需求，平均每秒25帧的画面渲染切换。

但是通常情况下，为了避免视频出现突然僵硬、卡顿的效果所需要的最低 FPS 为每秒 30 帧。所以系统的视频播放帧率目前是低于30，但考虑到系统所使用的测试数据并不是处理好的流媒体数据帧，而是在浏览器增加了解封装、解码原始数据这类耗时操作，所以可以认定系统对于音视频的播放实现上的性能损失在可接受范围之内。

## 4总结

通过以上研究实验证明通过迁移WebAssembly编码编译FFmpeg 开源音视频解码模块，使得以C语言编写的音视频解封装、解码等一系列CPU密集型操作可以在浏览器环境中正常调用执行，并有不错的运行速度。同时通过WebGL渲染引擎利用客户端的GPU加速提高视频图像的渲染性能。通过实验和系统实践，WebAssembly这种新型的二进制编码可以帮助浏览器弥补性能缺陷，让Web技术有了更多的可能与丰富的想象力，同时给迁移C/C++这种编译型语言库在Web浏览器中调用执行提供了理论验证和实践基础。

与此同时，本文内容中研究的音视频播放系统还存在以下几个缺点：首先是对客户端软件及硬件有苛刻的要求，必须要支持WebGL以及Web Worker。第二，当前音视频播放的快进、指定播放时间的功能还不够完善，存在操作后音视频不同步的问题。以上这些问题还有待进一步的研究与实践。

## 参考文献

[1]高萌. H.265视频编码相关技术研究[J]. 通讯世界,2019,26(4):17-18. DOI:10.3969/j.issn.1006-4222.2019.04.010.

[2]Scott Carey. The rise of WebAssembly[J]. InfoWorld.com,2022.

[3]王迪,高树论,蔡晓晰,丁钢波,冯雨润之,刘笑笑,王银.基于Canvas长连接浏览器播放H.265码流视频的应用研究[J].工业控制计算机,2021,34(08):99-100.

[4]Sven Groppe,Niklas Reimer. Code Generation for Big Data Processing in the Web using WebAssembly[J]. Open Journal of Cloud Computing,2019,6(1).

[5]Andreas Haas,Andreas Rossberg,Derek L. Schuff,Ben L. Titzer,Michael Holman,Dan Gohman,Luke Wagner,Alon Zakai,JF Bastien. Bringing the web up to speed with WebAssembly[P]. Programming Language Design and Implementation,2017.

[6]Olivera Solís, Rafael Alejandro,López Pérez, Yazdani. Codificación de video en HEVC/H.265 utilizando FFMPEG[J]. Ingeniería Electrónica, Automática y Comunicaciones,2019,40(2).

[7]岳瑞. 基于FFmpeg的音视频转码系统的设计与实现[D].西安电子科技大学,2021.

[8]谢贤博,聂芸,邓红艳,等. 基于WebGL的Canvas元素2D绘制加速[J]. 软件,2016,37(12):146-152. DOI:10.3969/j.issn.1003-6970.2016.12.031.

[9]薛超. 基于WebAssembly的JavaScript性能优化方案研究与实现[D].西北大学,2019.

[10]于航. 深入浅出WebAssembly[M]. 北京:电子工业出版社，2018

[11]才争野.基于FFmpeg与CUDA的YUV与RGB色彩空间转换性能对比分析[J].广东通信技术,2019,39(01):21-24.