基于 WebAssembly 和 WebGL 浏览器播放 H.265 编码视频的系统设计

缑通旺 [1]

(1.东南大学软件工程学院，苏州，215000;)

摘　要：针对大部分浏览器因为 H.265 编码技术许可协议的问题而不支持对 H.265 编码类型视频的直接播放，设计基于 WebAssembly 和 WebGL 的浏览器对 H.265 编码视频的播放系统。本文首先介绍系统的整体架构以及功能模块架构，并利用 FFmpeg 开源音视频解码库，结合迁移 WebAssembly 编码方案提高浏览器音视频解码速度。以及 WebGL 图像引擎渲染解码后的视频帧数据的方法。最终实现 Web 浏览器对 FFmpeg 解码库的调用执行，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[1]

(1. School of Software Engineering, Southeast University, SuZhou, 215000;)

Abstract: Aiming at the fact that most browsers do not support the direct playback of H.265 encoded videos due to the H.265 encoding technology license agreement, a WebAssembly and WebGL-based browser's playback system for H.265 encoded videos is designed. This paper first introduces the overall architecture and functional module architecture of the system, and uses the FFmpeg open source audio and video decoding library, combined with the migration of the WebAssembly encoding scheme to improve the browser audio and video decoding speed. And a method for the WebGL image engine to render the decoded video frame data. Finally, the Web browser can call and execute the FFmpeg decoding library, which improves the decoding speed and rendering performance, and optimizes the performance experience of video playback. The final experiment also shows that the video playback scheme of the system is feasible and the optimization effect is good. At the same time, it provides a theoretical verification and a practical basis for migrating the code base of CPU-intensive task processing to the browser.

Key words: 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之一。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。但在复杂的业务场景中需要浏览器承担一部分 CPU 密集型计算，所以在工业

作者简介：缑通旺，（1998-），男，硕士研究生，E-mail: gnoe@foxmail.com;

界为了让代码在浏览器中跑的更快，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（简称为 WASM），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]，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、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。针对以上问题，本文通过迁移 WASM 编码[4-5]对 FFmpeg 解码库编译[6-7]，并结合 WebGL 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[8]。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 编码视频结合 WASM 编码和 WebGL 渲染技术实现的视频播放系统架构如图 1 所示：

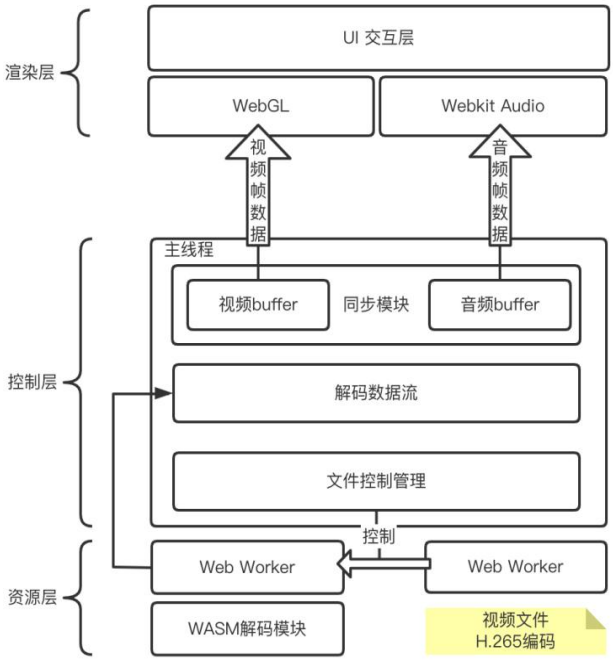

图 1 系统架构结构

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。

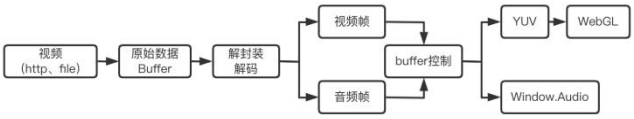

图 2 音视频解码播放流程

2 系统关键技术

2.1 基于 FFmpeg 二次开发模块

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、H.264、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[9]。FFmpeg 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 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

同步控制模块主要目的是实现音画同步消费的问题。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(1)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。

$$timestamp = pts * av\_q2d(time\_base) \quad (1)$$

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联系。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 编译 WASM 模块

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，它可以在现代网络浏览器中直接运行[10]。其拥有其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 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[11]。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，

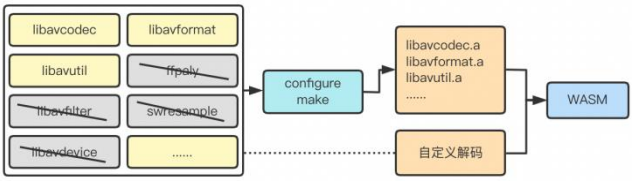

图 3 定制化编译流程模型

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”]

本小节通过对 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定义的方法的功能。至此，作为 H.265 编码的视频播放系统的基础核心已经全部实现。

2.3 基于 WebGL 引擎渲染帧图像

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[8]。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渲染加速，即 GPU 加速，使用 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[12]。

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 指代三种通道对画面上每一个像素点进行渲染。

YUV 颜色空间中，Y 指代明亮度、U 和 V 分别指代色度。因为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。

所以通常在传输、存储时按照 YUV 格式存储，在最后实际渲染的时候转换为 RGB 格式对每一个像素点进行渲染。所以需要在处理播放时需要对颜色数据格式进行转换，转互转换的公式如下所示

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad (2)$$

2.3.2 WebGL 绘制 YUV420

YUV420 存储格式首先将二维的图像数据转换为一维数据通过数据进行存储，先存储所有的 Y 分量数据，再存储 UV 分量数据，并且 YUV 各个分量按照 4:1:1 的比例进行分配。如图 4 所示，也就是说 Y1、Y2、Y7、Y8 都分别共用了 U1、V1 分量。通过这种存储格式可以极大的压缩原音视频数据。最后本系统通过读取帧画面解析出来的数据存放在一个长数组中，根据图像大小比例，可以得出 U 分量、V 分量的偏移位置，从而进行数据读取渲染。

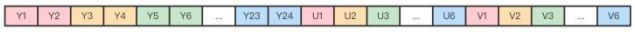

图 4 YUV420 存储格式

在绘制一帧画面的时候，对 Y、U、V 三个分量数据，分别初始化一个 WebGL 画笔为 Y-GL、U-GL、V-GL。依次读取数组中各个分量的数据，先用 Y-GL 绘制 Y 分量的数据，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 画笔绘制 U 分量数据，最后根据 U 和 V 分量数据大小以及偏移量最后绘制 V 分量数据。

3 实验

实验硬件：Mac 2GHz 四核 Intel Core i5，图形卡：Intel Iris Plus Graphics 1536MB。

运行环境：Google 浏览器 x86_64 version；Node 16.3.0 version。

3.1 WebGL 和 Canvas 的性能对比实验

现代浏览器为 Canvas 2D 和 WebGL 都提供硬件加速的支持[8]。本小节针对 WebGL 和 Canvas 2D 渲染不同数量梯度的粒子图像进行对比实验，其中每一个粒子是宽 2 像素、高 2 像素的黑色点图像，再分别使用 WebGL 和 Canvas 2D 绘制 1000、5000、10000 数量级的粒子图像模拟绘制视频图像的情况，并查看浏览器绘制页面时的 FPS（Frame Per Second）波动情况，可以分析随着粒子图像数量的增多，WebGL 和 Canvas 2D 在渲染上的表现情况。最终选取了实验过程中具有代表性的 Canvas 2D 渲染 1000 粒子如图 5、渲染 10000 粒子如图 6 和 WebGL 渲染 10000 粒子如图 7 的具体渲染情况所示。

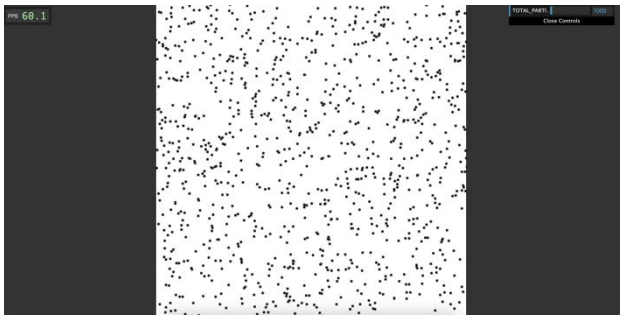

图 5 Canvas 绘制 1000 粒子

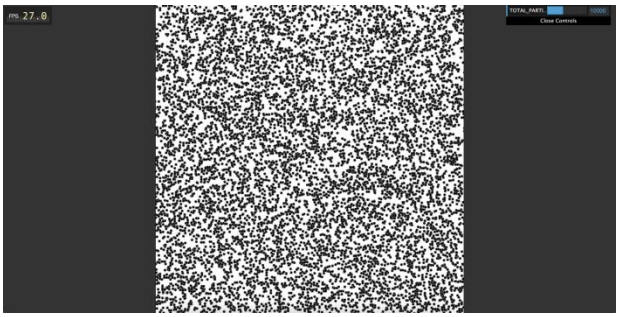

图 6 Canvas 绘制 10000 粒子

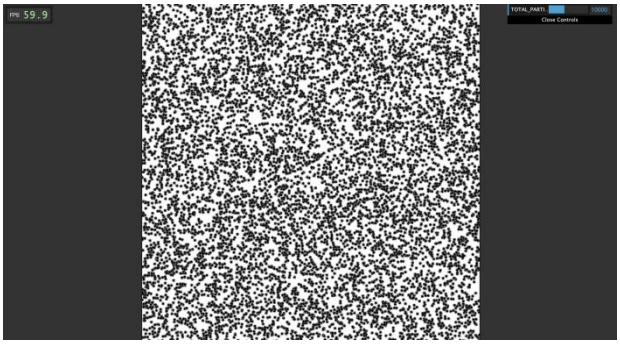

图 7 WebGL 绘制 10000 粒子

从上图可以清楚的看到 Canvas 在渲染粒子图像数量增多时其 FPS 数值在降低，FPS 代表当前绘制页面每秒的帧率，当帧率越大说明页面粒子绘制的性能越高、速度越快、越稳定，反之越差。最终实验结果汇总如表 2 所示：

表 2 Canvas 和 WebGL 渲染粒子图像的表现情况

数量	1000	5000	10000
Canvas	60.1fps	53.5fps	27.0fps
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 WASM 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 函数运行实验

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 执行引擎来执行 node hello.js 命令，最终无论是浏览器控制台还是 Node 环境中测试，可以得出通过 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。

3.2.2 斐波那契数列函数测试实验

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。分别编写 JavaScript 版本和 C 语言版本的斐波那契数列函数代码，并通过 emcc 命令将 fib.c 文件单独编译为 WASM 模块，通过如图 8 的 JavaScript 调用程序对 WASM 模块引入调用。

```
// wasm模块的运行测试
fetch('./fib.wasm')
  .then((res) => {
    return res.arrayBuffer()
  })
  .then(WebAssembly.instantiate)
  .then((module) => {

    [20, 40, 45].forEach(n => {
      const start = performance.now();
      module.instance.exports.fib(45);
      const end = performance.now();
    });
  });
```

图 8 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 3 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。

3.3 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，选取 25.6MB 大小的 H.265 编码视频进行解码播放测试，打开 localhost:8080 查看视频播放页面。

计算视频播放的码率公式为：

$$rate = video\_size \div t \tag{3}$$

视频大小为 25.6MB，播放时间长度共 467s，可得系统播放视频的码率为 438.5kbps。

同时计算每一帧图像的大小计算公式为：

$$size = W \times H \times pix\_fmt \tag{4}$$

视频播放页面像素大小为宽 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。同时使用浏览器的开发者工具对视频播放系统进行性能指标的相关测试。

如图 9 所示是浏览器播放视频时测试的性能火焰图，可以发现在处理一帧数据时 displayAudioFrame 音频播放和 displayVideoFrame 图像渲染分别耗时为 0.2ms 和 0.3ms。

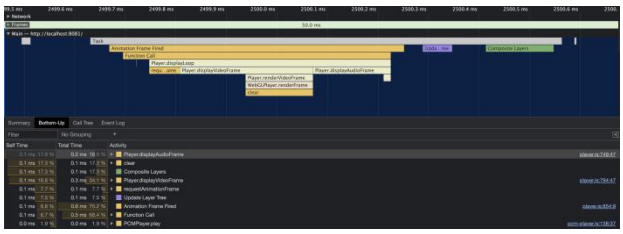

图 9 浏览器播放视频性能火焰图

如图 10 所示, 是针对系统播放时的内存使用情况的指标信息，可以发现整体上视频播放系统的 CPU 使用、JavaScript 堆栈内存以及 DOM 节点的波动都很稳定，其中 CPU 使用率在 3%～7.5%波动，可以说明本视频播放系统的设计方案是稳定可行的，只有刚开始初始化加载页面时达到各项指标的峰值。

图 10 Memory Inspector

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。经测试总体上本系统视频播放平均为每秒 25 帧的帧率画面切换, 其音视频播放的稳定性和流畅度以及帧率、码率、渲染耗时、内存使用情况等各项指标均符合预期。

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

通过以上实验证明通过迁移 WASM 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, WASM 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，本文设计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、Web Worker。第二，当前音视频播放快进、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等问题还需解决。

参考文献:

[1] 高萌. H.264 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，蔡晓晰等.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, et al. Bringing the web up to speed with WebAssembly[P]. Programming Language Design and Implementation, 2017.

[6] Olivera Solís, Rafael Alejandro, López Pérez, et al. 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] 宋燕燕，秦军. 基于 FFMPEG 的视频水印系统设计[J]. 计算机技术与发展, 2018, 28(4): 4.

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

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

[12] 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): 2124.