Extending the browser with WebAssembly

- Choosing an interesting code-base
- Adapting for use in the browser
  - Using cmake to build the source code
  - Designing an API to interface to the library
- Building a test harness to test outside the browser
- Implementing a buffering mechanism for multiple video frames
- Getting the video frames onto the page with WebGL
- Future work and takeaways
- Credits

One of the best things about WebAssembly is the ability experiment with new capabilities and implement new ideas before the browser ships those features natively (if at all). You can think of using WebAssembly this way as a high-performance polyfill mechanism, where you write your feature in C/C++ or Rust rather than JavaScript.

With a plethora of existing code available for porting, it's possible to do things in the browser that weren't viable until WebAssembly came along.

This article will walk through an example of how to take the existing AV1 video codec source code, build a wrapper for it, and try it out inside your browser and tips to help with building a test harness to debug the wrapper. Full source code for the example here is available at github.com/GoogleChromeLabs/wasm-av1 for reference.

Download one of these two 24fps test video files and try them on our built demo.

Choosing an interesting code-base

For a number of years now, we've seen that a large percentage of traffic on the web consists of video data, Cisco estimates it as much as 80% in fact! Of course, browser vendors and video sites are hugely aware of the desire to reduce the data consumed by all this video content. The key to that, of course, is better compression, and as you'd expect there is a lot of research into next-generation video compression aimed at reducing the data burden of shipping video across the internet.

As it happens, the Alliance for Open Media has been working on a next generation video compression scheme called AV1 that promises to shrink video data size considerably. In the future, we'd expect browsers to ship native support for AV1, but luckily the source code for the compressor and decompressor are open source, which makes that an ideal candidate for trying to compile it into WebAssembly so we can experiment with it in the browser.

> {% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/Nxc2in5sTnWc8T1slk0h.png", alt="Bunny movie image.", width="360", height="240" %}

Adapting for use in the browser

One of the first things we need to do to get this code into the browser is to get to know the existing code to understand what the API is like. When first looking at this code, two things stand out:

1. The source tree is built using a tool called `cmake`; and
2. There are a number of examples that all assume some kind of file-based interface.

All the examples that get built by default can be run on the command line, and that is likely to be true in many other code bases available in the community. So, the interface we're going to build to make it run in the browser could be useful for many other command line tools.

Using `cmake` to build the source code

Fortunately, the AV1 authors have been experimenting with Emscripten, the SDK we're going to use to build our WebAssembly version. In the root of the AV1 repository, the file `CMakeLists.txt` contains these build rules:

```
if(EMSCRIPTEN)
add_preproc_definition(_POSIX_SOURCE)
append_link_flag_to_target("inspect" "-s TOTAL_MEMORY=402653184")
append_link_flag_to_target("inspect" "-s MODULARIZE=1")
append_link_flag_to_target("inspect"
                           "-s EXPORT_NAME=\"\'DecoderModule\'\"")
```

```
append_link_flag_to_target("inspect" "--memory-init-file 0")

if("${CMAKE_BUILD_TYPE}" STREQUAL "")
    # Default to -O3 when no build type is specified.
    append_compiler_flag("-O3")
endif()
em_link_post_js(inspect "${AOM_ROOT}/tools/inspect-post.js")
endif()
```

The Emscripten toolchain can generate output in two formats, one is called `asm.js` and the other is WebAssembly. We'll be targeting WebAssembly as it produces smaller output and can run faster. These existing build rules are meant to compile an `asm.js` version of the library for use in an inspector application that's leveraged to look at the content of a video file. For our usage, we need WebAssembly output so we add these lines just before the closing `endif()` statement in the rules above.

```
# Force generation of Wasm instead of asm.js
append_link_flag_to_target("inspect" "-s WASM=1")
append_compiler_flag("-s WASM=1")
```

{% Aside %} It's important to create a build directory that's separate from the source code tree, and run all the commands below inside that build directory. {% endAside %}

Building with `cmake` means first generating some `Makefiles` by running `cmake` itself, followed by running the command `make` which will perform the compilation step. Note, that since we are using Emscripten we need to use the Emscripten compiler toolchain rather than the default host compiler. That's achieved by using `Emscripten.cmake` which is part of the Emscripten SDK and passing it's path as a parameter to `cmake` itself. The command line below is what we use to generate the Makefiles:

```
cmake path/to/aom \
  -DENABLE_CCACHE=1 -DAOM_TARGET_CPU=generic -DENABLE_DOCS=0 \
  -DCONFIG_ACCOUNTING=1 -DCONFIG_INSPECTION=1 -DCONFIG_MULTITHREAD=0 \
  -DCONFIG_RUNTIME_CPU_DETECT=0 -DCONFIG_UNIT_TESTS=0
  -DCONFIG_WEBM_IO=0 \
  -DCMAKE_TOOLCHAIN_FILE=path/to/emsdk-portable/.../Emscripten.cmake
```

The parameter `path/to/aom` should be set to the full path of the location of the AV1 library source files. The `path/to/emsdk-portable/…/Emscripten.cmake` parameter needs to be set to the path for the Emscripten.cmake toolchain description file.

For convenience we use a shell script to locate that file:

```
#!/bin/sh
EMCC_LOC=`which emcc`
EMSDK_LOC=`echo $EMCC_LOC | sed 's?/emscripten/[0-9.]*/emcc??'`
EMCMAKE_LOC=`find $EMSDK_LOC -name Emscripten.cmake -print`
echo $EMCMAKE_LOC
```

If you look at the top-level `Makefile` for this project, you can see how that script is used to configure the build.

Now that all of the setup has been done, we simply call `make` which will build the entire source tree, including samples, but most importantly generate `libaom.a` which contains the video decoder compiled and ready for us to incorporate into our project.

Designing an API to interface to the library

Once we've built our library, we need to work out how to interface with it to send compressed video data to it and then read back frames of video that we can display in the browser.

Taking a look inside the AV1 code tree, a good starting point is an example video decoder which can be found in the file `[simple_decoder.c](https://aomedia.googlesource.com/aom/+/master/examples/simple_decoder.c)`. That decoder reads in an IVF file and decodes it into a series of images that represent the frames in the video.

We implement our interface in the source file `[decode-av1.c](https://github.com/GoogleChromeLabs/wasm-av1/blob/master/decode-av1.c)`.

Since our browser can't read files from the file system, we need to design some form of interface that lets us abstract away our I/O so that we can build something similar to the example decoder to get data into our AV1 library.

On the command line, file I/O is what's known as a stream interface, so we can just define our own interface that looks like stream I/O and build whatever we like in the underlying implementation.

We define our interface as this:

```
DATA_Source *DS_open(const char *what);
size_t      DS_read(DATA_Source *ds,
                    unsigned char *buf, size_t bytes);
int         DS_empty(DATA_Source *ds);
void        DS_close(DATA_Source *ds);
// Helper function for blob support
void        DS_set_blob(DATA_Source *ds, void *buf, size_t len);
```

The `open/read/empty/close` functions look a lot like normal file I/O operations which allows us to map them easily onto file I/O for a command line application, or implement them some other way when run inside a browser. The `DATA_Source` type is opaque from the JavaScript side, and just serves to encapsulate the interface. Note, that building an API that closely follows file semantics makes it easy to reuse in many other code-bases that are intended to be used from a command line (e.g. diff, sed, etc.).

We also need to define a helper function called `DS_set_blob` that binds raw binary data to our stream I/O functions. This lets the blob be 'read' as if it's a stream (i.e. looking like a sequentially read file).

Our example implementation enables reading the passed in blob as if it was a sequentially read data source. The reference code can be found in the file `blob-api.c`, and the entire implementation is just this:

```
struct DATA_Source {
    void        *ds_Buf;
    size_t      ds_Len;
    size_t      ds_Pos;
};

DATA_Source *
DS_open(const char *what) {
    DATA_Source     *ds;

    ds = malloc(sizeof *ds);
    if (ds != NULL) {
        memset(ds, 0, sizeof *ds);
    }
    return ds;
}

size_t
DS_read(DATA_Source *ds, unsigned char *buf, size_t bytes) {
    if (DS_empty(ds) || buf == NULL) {
        return 0;
    }
    if (bytes > (ds->ds_Len - ds->ds_Pos)) {
        bytes = ds->ds_Len - ds->ds_Pos;
    }
    memcpy(buf, &ds->ds_Buf[ds->ds_Pos], bytes);
```

```
    ds->ds_Pos += bytes;

    return bytes;
}

int
DS_empty(DATA_Source *ds) {
    return ds->ds_Pos >= ds->ds_Len;
}

void
DS_close(DATA_Source *ds) {
    free(ds);
}

void
DS_set_blob(DATA_Source *ds, void *buf, size_t len) {
    ds->ds_Buf = buf;
    ds->ds_Len = len;
    ds->ds_Pos = 0;
}
```

Building a test harness to test outside the browser

One of the best practices in software engineering is to build unit tests for code in conjunction with integration tests.

When building with WebAssembly in the browser, it makes sense to build some form of unit test for the interface to the code we're working with so we can debug outside of the browser and also be able to test out the interface we've built.

In this example we've been emulating a stream based API as the interface to the AV1 library. So, logically it makes sense to build a test harness that we can use to build a version of our API that runs on the command line and does actual file I/O under the hood by implementing the file I/O itself underneath our `DATA_Source` API.

The stream I/O code for our test harness is straightforward, and looks like this:

```
DATA_Source *
DS_open(const char *what) {
    return (DATA_Source *)fopen(what, "rb");
}

size_t
DS_read(DATA_Source *ds, unsigned char *buf, size_t bytes) {
    return fread(buf, 1, bytes, (FILE *)ds);
}

int
DS_empty(DATA_Source *ds) {
    return feof((FILE *)ds);
}

void
DS_close(DATA_Source *ds) {
    fclose((FILE *)ds);
}
```

By abstracting the stream interface we can build our WebAssembly module to use binary data blobs when in the browser, and interface to real files when we build the code to test from the command line. Our test harness code can be found in the example source file `test.c`.

Implementing a buffering mechanism for multiple video frames

When playing back video, it's common practice to buffer a few frames to help with smoother playback. For our purposes we'll just implement a buffer of 10 frames of video, so we'll buffer 10 frames before we start playback. Then each time a frame is displayed, we'll try to decode another frame so we keep the buffer full. This approach makes sure frames are available in advance to help stop the video stuttering.

With our simple example, the entire compressed video is available to read, so the buffering isn't really needed. However, if we're to extend the source data interface to support streaming input from a server, then we need to have the buffering mechanism in place.

The code in `decode-av1.c` for reading frames of video data from the AV1 library and storing in the buffer as this:

```
void
AVX_Decoder_run(AVX_Decoder *ad) {
    ...
    // Try to decode an image from the compressed stream, and buffer
    while (ad->ad_NumBuffered < NUM_FRAMES_BUFFERED) {
        ad->ad_Image = aom_codec_get_frame(&ad->ad_Codec,
                                           &ad->ad_Iterator);
        if (ad->ad_Image == NULL) {
            break;
        }
        else {
            buffer_frame(ad);
        }
    }
```

We've chosen to make the buffer contain 10 frames of video, which is just an arbitrary choice. Buffering more frames means more waiting time for the video to begin playback, whilst buffering too few frames can cause stalling during playback. In a native browser implementation, buffering of frames is far more complex than this implementation.

Getting the video frames onto the page with WebGL

The frames of video that we've buffered need to be displayed on our page. Since this is dynamic video content, we want to be able to do that as fast as possible. For that, we turn to WebGL.

WebGL lets us take an image, such as a frame of video, and use it as a texture that gets painted on to some geometry. In the WebGL world, everything consists of triangles. So, for our case we can use a convenient built in feature of WebGL, called gl.TRIANGLE_FAN.

However, there is a minor problem. WebGL textures are supposed to be RGB images, one byte per color channel. The output from our AV1 decoder is images in a so-called YUV format, where the default output has 16 bits per channel, and also each U or V value corresponds to 4 pixels in the actual output image. This all means we need to color convert the image before we can pass it to

WebGL for display.

To do so, we implement a function `AVX_YUV_to_RGB()` which you can find in the source file `yuv-to-rgb.c`. That function converts the output from the AV1 decoder into something we can pass to WebGL. Note, that when we call this function from JavaScript we need to make sure that the memory we're writing the converted image into has been allocated inside the WebAssembly module's memory - otherwise it can't get access to it. The function to get an image out from the WebAssembly module and paint it to the screen is this:

```
function show_frame(af) {
    if (rgb_image != 0) {
        // Convert The 16-bit YUV to 8-bit RGB
        let buf = Module._AVX_Video_Frame_get_buffer(af);
        Module._AVX_YUV_to_RGB(rgb_image, buf, WIDTH, HEIGHT);
        // Paint the image onto the canvas
        drawImageToCanvas(new Uint8Array(Module.HEAPU8.buffer,
                rgb_image, 3 * WIDTH * HEIGHT), WIDTH, HEIGHT);
    }
}
```

The `drawImageToCanvas()` function that implements the WebGL painting can be found in the source file `draw-image.js` for reference.

Future work and takeaways

Trying our demo out on two test video files (recorded as 24 f.p.s. video) teaches us a few things:

1. It's entirely feasible to build a complex code-base to run performantly in the browser using WebAssembly; and
2. Something as CPU intensive as advanced video decoding is feasible via WebAssembly.

There are some limitations though: the implementation is all running on the main thread and we interleave painting and video decoding on that single thread. Offloading the decoding into a web worker could provide us with smoother playback, as the time to decode frames is highly dependent on the content of that frame and can sometimes take more time than we have budgeted.

The compilation into WebAssembly uses the AV1 configuration for a generic CPU type. If we compile natively on the command line for a generic CPU we see similar CPU load to decode the video as with the WebAssembly version, however the AV1 decoder library also includes SIMD implementations that run up to 5 times faster. The WebAssembly Community Group is currently working on extending the standard to include SIMD primitives, and when that comes along it

promises to speed up decoding considerably. When that happens, it'll be entirely feasible to decode 4k HD video in real-time from a WebAssembly video decoder.

In any case, the example code is useful as a guide to help port any existing command line utility to run as a WebAssembly module and shows what's possible on the web already today.

Credits

Thanks to Jeff Posnick, Eric Bidelman and Thomas Steiner for providing valuable review and feedback.

□□□ #2
□ □□□ □□□ 2026-10-31 22:25:14 UTC
□ □□□ □□□ 2026-10-31 22:25:15 UTC