Asm.js and Web Assembly: Speeding Up The Web

I've been hearing about Web Assembly and its predecessor, ASM.js for a while. The idea is that we can bring C and C++ code into the web and use it directly on the browser without having plugins get in the way. This would also make it easier to port high end games and other C/C++ code to Javascript and leverage existing APIs and features

Asm.js

> Asm.js is a subset of JavaScript that is heavily restricted in what it can do and how it can operate. This is done so that the compiled Asm.js code can run as fast as possible making as few assumptions as it can, converting the Asm.js code directly into assembly. It's important to note that Asm.js is just JavaScript – there is no special browser plugin or feature needed in order to make it work (although a browser that is able to detect and optimize Asm.js code will certainly run faster). It's a specialized subset of JavaScript that's optimized for performance, especially for this use case of applications compiled to JavaScript.
>
> — Asm.js: The JavaScript Compile Target

The first attempt at using Javascript as a target language for cross compilation is asm.js. Using Emscripten asm.js allowed developers to compile massive C/C++ code bases to Javascript that that ran natively in the browser and leverages web technologies and APIs like WebGL being able to port games created with Unity and Unreal engine directly to the web like the Unreal demo below, circa 2013

The process can be illustrated with the diagram below (taken from ejohn.org):

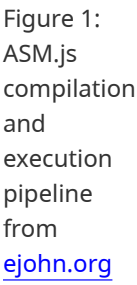

Figure 1: ASM.js compilation and execution pipeline from ejohn.org

The code is not meant to be written, or read, by humans. The example below was created by John Ressig to demonstrate the differences between asm.js and the regular Javascript code developers normally work with. The code has been formatted for clarity and sanity preservation, standard asm.js is heavily minified into one continuous blog of text.

```
function Vb(d) {
    d = d | 0;
    var e = 0, f = 0, h = 0, j = 0, k = 0, l = 0, m = 0, n = 0,
        o = 0, p = 0, q = 0, r = 0, s = 0;
    e = i;
    i = i + 12 | 0;
    f = e | 0;
    h = d + 12 | 0;
    j = c[h >> 2] | 0;
```

```
    if ((j | 0) > 0) {
        c[h >> 2] = 0;
        k = 0
    } else {
        k = j
    }
    j = d + 24 | 0;
    if ((c[j >> 2] | 0) > 0) {
        c[j >> 2] = 0
    }
    l = d + 28 | 0;
    c[l >> 2] = 0;
    c[l + 4 >> 2] = 0;
    l = (c[1384465] | 0) + 3 | 0;
    do {
        if (l >>> 0 < 26) {
            if ((4980736 >>> (l >>> 0) & 1 | 0) == 0) {
                break
            }
            if ((c[1356579] | 0) > 0) {
                m = d + 4 | 0;
                n = 0;
                while (1) {
                    o = c[(c[1356577] | 0) + (n << 2) >> 2] | 0;
                    do {
                        if (a[o + 22 | 0] << 24 >> 24 == 24) {
                            if (!(Vp(d, o | 0) | 0)) {
                                break
                            }
                            p = (c[m >> 2] | 0) + (((c[h >> 2] | 0) - 1 |
                            q = o + 28 | 0;
                            c[p >> 2] = c[q >> 2] | 0;
                            c[p + 4 >> 2] = c[q + 4 >> 2] | 0;
                            c[p + 8 >> 2] = c[q + 8 >> 2] | 0;
                            c[p + 12 >> 2] = c[q + 12 >> 2] | 0;
                            c[p + 16 >> 2] = c[q + 16 >> 2] | 0;
                            c[p + 20 >> 2] = c[q + 20 >> 2] | 0;
                            c[p + 24 >> 2] = c[q + 24 >> 2] | 0
```

```
                }
            } while (0);
            o = n + 1 | 0;
            if ((o | 0) < (c[1356579] | 0)) {
                n = o
            } else {
                break
            }
        }
        r = c[h >> 2] | 0
    } else {
        r = k
    } if ((r | 0) == 0) {
        i = e;
        return
    }
    n = c[j >> 2] | 0;
    if ((n | 0) >= 1) {
        i = e;
        return
    }
    m = f | 0;
    o = f + 4 | 0;
    q = f + 8 | 0;
    p = n;
    while (1) {
        g[m >> 2] = 0.0;
        g[o >> 2] = 0.0;
        g[q >> 2] = 0.0;
        Vq(d, p, f, 0, -1e3);
        n = c[j >> 2] | 0;
        if ((n | 0) < 1) {
            p = n
        } else {
            break
        }
    }
    i = e;
```

```
            return
        }
    } while (0);
    if ((c[1356579] | 0) <= 0) {
        i = e;
        return
    }
    f = d + 16 | 0;
    r = 0;
    while (1) {
        k = c[(c[1356577] | 0) + (r << 2) >> 2] | 0;
        do {
            if (a[k + 22 | 0] << 24 >> 24 == 30) {
                h = b[k + 14 >> 1] | 0;
                if ((h - 1 & 65535) > 1) {
                    break
                }
                l = c[j >> 2] | 0;
                p = (c[1384465] | 0) + 3 | 0;
                if (p >>> 0 < 26) {
                    s = (2293760 >>> (p >>> 0) & 1 | 0) != 0 ? 0 : -1e3
                } else {
                    s = -1e3
                } if (!(Vq(d, l, k | 0, h << 16 >> 16, s) | 0)) {
                    break
                }
                g[(c[f >> 2] | 0) + (l * 112 & -1) + 56 >> 2] = +(b[k + 12
                h = (c[f >> 2] | 0) + (l * 112 & -1) + 60 | 0;
                l = k + 28 | 0;
                c[h >> 2] = c[l >> 2] | 0;
                c[h + 4 >> 2] = c[l + 4 >> 2] | 0;
                c[h + 8 >> 2] = c[l + 8 >> 2] | 0;
                c[h + 12 >> 2] = c[l + 12 >> 2] | 0;
                c[h + 16 >> 2] = c[l + 16 >> 2] | 0;
                c[h + 20 >> 2] = c[l + 20 >> 2] | 0;
                c[h + 24 >> 2] = c[l + 24 >> 2] | 0
            }
        } while (0);
```

```
        k = r + 1 | 0;
        if ((k | 0) < (c[1356579] | 0)) {
            r = k
        } else {
            break
        }
    }
    i = e;
    return
}
```

Handwritten asm.js code is marginally easier to understand. the example below, taken from the asm.js specification shows what it looks like when we write asm.js code by hand

```
function DiagModule(stdlib, foreign, heap) {
  "use asm";

  // Variable Declarations
  var sqrt = stdlib.Math.sqrt;

  // Function Declarations
  function square(x) {
      x = +x;
      return +(x*x);
  }

  function diag(x, y) {
      x = +x;
      y = +y;
      return +sqrt(square(x) + square(y));
  }

  return { diag: diag };
}
```

Am asm.js module is contained within a function and starts with the `"use asm";` directive at the top. This tells the interpreter that everything inside the function

should be handled as asm.js and be compiled to assembly directly without going through the regular Javascript interpreter / optimization cycles.

Note the three arguments for the asm.js function: `stdlib`, `foreign`, and heap.

- The stdlib object contains references to a number of built-in math functions
- foreign provides access to custom user-defined functionality, such as drawing a shape in WebGL
- heap gives you an ArrayBuffer which can be viewed through a number of different lenses, such as Int32Array and Float32Array.

The rest of the module is broken up into three parts: variable declarations, function declarations, and finally an object exporting the functions to expose to the user.

The export is an essential point to understand. It allows all of the code within the module to be handled as asm.js but still be available to other, normal, JavaScript code. You could, theoretically, have some code that looks like the following, using the above DiagModule code:

```
document.body.onclick = function() {
  function DiagModule(stdlib){"use asm"; ... return { ... };}

  var diag = DiagModule({ Math: Math }).diag;
  alert(diag(10, 100));
};
```

This would result in an asm.js DiagModule that's handled special by the JavaScript interpreter but still made available to other JavaScript code that could still access it and use it within a click handler.

The result is that, within limits, we can bring C and C++ content directly into the web platform. Games and other large codebases written in C and C++ can be ported to work on the web. Mozilla Hacks' Porting to Emscripten tells of one project's migration to asm.js.

Web Assembly

Web Assembly is an evolution of asm.js that has taken a lot of the lessons browser

vendors and implementors learned from working with asm.js. The Web Assembly generated code is binary rather than text-based but can still provide two-way interaction with native Javascript code.

Installing Emscripten SDK

To compile C/C++ code to web assembly you need to have the Emscripten SDK installed on your system. I will only cover installing the portable SDK as it works across platforms and doesn't require administrator privileges in any platform (Windoes, Mac or Linud). For other installation methods check

The Portable Emscripten SDK is a no-installer version of the SDK package. It is identical to the NSIS installer, except that it does not interact with the Windows registry. This allows Emscripten to be used on a computer without administrative privileges, and means that the installation can be migrated from one location (directory or computer) to another by simply copying the directory contents to the new location.

First check the Platform-specific notes below (or online) and install any prerequisites.

Install or update the SDK using the following steps:

Download and unzip the portable SDK package to a directory of your choice. This directory will contain the Emscripten SDK.

Open a command prompt inside the SDK directory and run the following emsdk commands to get the latest tools from Github and set them as active:

```
# Fetch the latest registry of available tools.
./emsdk update

# Download and install the latest SDK tools.
./emsdk install latest

# Make the "latest" SDK "active"
./emsdk activate latest
```

Notes:

On Windows, invoke the tool with emsdk instead of ./emsdk.

Linux and Mac OS X only: Call source ./emsdk_env.sh after activate to set the system path to the active version of Emscripten.

Platform-specific notes

Mac OS X

These instructions explain how to install all the required tools. You can test whether some of these are already installed on the platform and skip those steps.

1. Install the XCode Command Line Tools. These are a precondition for git.
    1. Install XCode from the Mac OS X App Store.
    2. In XCode | Preferences | Downloads, install Command Line Tools.
2. Install git:
    1. Allow installation of unsigned packages, or installing the git package won't succeed.
    2. Install XCode and the XCode Command Line Tools (should already have been done). This will provide git to the system PATH (see this stackoverflow post)
    3. Download and install git directly from http://git-scm.com/
3. Install cmake if you do not have it yet:
    1. Download and install latest CMake from Kitware CMake downloads
    2. Install node.js from nodejs.org

Linux

Pre-built binaries of tools are not available on Linux. Installing a tool will automatically clone and build that tool from the sources inside the emsdk directory.

> Emsdk does not install any tools to the system, or otherwise interact with Linux package managers. All file changes are done inside the **emsdk/** directory.

Set the current Emscripten path on Linux/Mac OS X

```
source ./emsdk_env.sh
```

This step is not required on Windows because calling the activate command

also sets the correct system path (this is not possible on Linux due to security restrictions).

Whenever you change the location of the Portable SDK (e.g. take it to another computer), re-run the `./emsdk activate latest` command (and `source ./emsdk_env.sh` for Linux).

Compiling an application

Now that we have the portable SDK installed we can begin working on compiling code.

We'll run two examples. The first one will print `hello, world`. The second example is more complex and will produce a gradient square in WebGL, using the SDL library on the C side.

For each example we'll compile the code and generate a webpage to make sure the code works.

```
#include <stdio.h>

int main() {
  printf("hello, world!\n");
  return 0;
}
```

To compile the code and generate the web page associated with it the command to run is:

```
./emcc tests/hello_world.c -o hello_word.html
```

I'm running the command from within the Emscripten directory. Adjust your path as needed.

```
#include <stdio.h>
#include <SDL/SDL.h>
```

```
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#endif

extern "C" int main(int argc, char** argv) {
  printf("hello, world!\n");

  SDL_Init(SDL_INIT_VIDEO"C"  SDL_Surface *screen = SDL_SetVideoMode(256,

#ifdef TEST_SDL_LOCK_OPTS
  EM_ASM("SDL.defaults.copyOnLock = false; SDL.defaults.discardOnLock = tr
"SDL.defaults.copyOnLock = false; SDL.defaults.discardOnLock = true; SDL.

  if (SDL_MUSTLOCK(screen)) SDL_LockSurface(screen);
  for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
#ifdef TEST_SDL_LOCK_OPTS
      // Alpha behaves like in the browser, so write proper opaque pixels
      int alpha = 255;
#else
      // To emulate native behavior with blitting to screen, alpha compone
      // is ignored. Test that it is so by outputting data (and testing
      // that it does get discarded)
      int alpha = (i+j) % 255;
#endif  if (SDL_MUSTLOCK(screen)) SDL_UnlockSurface(screen);
  SDL_Flip(screen);

  printf("you should see a smoothly-colored square - no sharp lines but th
  printf("and here is some text that should be HTML-friendly: amp: |&| dou

  SDL_Quit();

  return 0;
}
"you should see a smoothly-colored square - no sharp lines but the square
```

The second demo works the same way. The code is more complex than the `hello_world` example and serves as an example of what you can do with the

technology, incorporating additional libraries and outputting to WebGL.

To run the compiler run the following command from the root of your Emscripten SDK:

```
./emcc tests/hello_world_sdl.cpp -o hello2.htm
```

So what do we use asm.js Web Assembly for?

Let me start by stating this very clearly: **asm.js and Web Assembly are not replacements for Javascript**. They provide direct access to the underlying C libraries and functionality and are usually faster than equivalent code in Javascript.

THey also bring large codebases to the browser without needing plugins or runtime environments. Unity used to have a plugin that users must install before running any Unity content. The plugin is no longer supported by the browsers and Unity games are moving to browser-based experiences. This becomes essential in mobile where installing apps is less attractive than just downloading content.

As Web Assembly matures I expect to see mixed libraries where most of the code is written in Javascript and the computationally expensive code (video compression and decompression, cryptography and others) are written in C, C++ or any other language supported by Web Assembly tools.

Exciting times indeed!