Computer Games Technology

A JavaScript Runtime for Hardware Accelerated Applications

Computing Honours Project (COMP10034) Interim Report

William Taylor

B00235610

06/01/2017

Supervisor: Paul Keir

Abstract

To investigate and explore how we can make hardware acceleration experimentation more accessible, a prototype runtime was developed on top of Google's V8 JavaScript compiler and popular open source frameworks. Reasons for creating the platform are numerous, they include making the technology more accessible and aiding pre-existing efforts to find new ways of leveraging the hardware. The development of the runtime is discussed in detail as are revisions to the JavaScript standard that allow it to be used for general-purpose scripting. The runtime demonstrates that an all in one platform can streamline development and make general purpose computation on graphics hardware easier for both novices and experts.

***Keywords**: V8, JavaScript, GPUs, GPGPU, OpenCL, OpenGL, CUDA, DirectX*

Contents

1.0 Introduction

1.1 Topic Overview

In recent history, there has been a seismic shift in technology. Processors have stopped getting faster at an exponential rate. Increasing the clock speed of processors has now been abandoned in favour of multicore processors (David Geer, 2005). Due to 3D and high resolution media increasing in popularity we now see Graphics Processing Units (GPUs) integrated into modern computers by default (Intelcom, 2016, Amdcom, 2016). Easily learning and experimenting with this new technology is of great importance if we are to see general purpose computing on graphics processing units (GPGPU) more widely adopted. This is the topic for this honours project where we will explore the possibility of an integrated platform for GPU technique experimentation and development. Specifically, we will look at a dedicated platform that leverages the popular scripting language JavaScript to provide a reliable and flexible tool to those learning how to leverage GPUs for the first time and to those who wish to develop their own GPU techniques in an easier manner.

1.2 The Problem

There are currently several problems we have identified in experimenting with GPUs and writing GPU based applications that limit a programmer's ability to get stuck into this exciting piece of technology.

One is that at the time of writing this paper there is currently no easy to use integrated environment to experiment with various GPU APIs such as OpenCL and OpenGL. While one could argue that the Web provides an integrated environment through WebCL and WebGL which are web equivalents of OpenCL and OpenGL, I would strongly disagree for several reasons. The first being that due to the requirement of a browser being portable it is unable to provide support for GPU technologies designed for specific hardware such as CUDA or specific APIs locked into a single operating system such as DirectX. The other reason for disagreeing is because the browser has a security model that disables local access to the computer making the loading of data such as complex 3D geometric models overbearing and complicated. So, while the Web may provide a way to write GPU programs it is more for web developers to speed up their applications and not to provide a toolset to make GPU programming as easy as possible.

Another issue identified is the frustration of using the native bindings to the APIs from C++. As C++ doesn't provide native support through the standard template library (STL) for images, models, input and windows it leads to a lot of extra work with additional libraries and APIs rather than letting you get on with your GPU technique development. The result is a lot of boilerplate before you get to writing what you will be experimenting with and that is the GPU programs themselves whether that is kernels in OpenCL or shaders in OpenGL. This issue is after you install various SDKs and tools to get access to these APIs, making it not only difficult when you start writing your program but difficult to get started in the first place.

1.3 The Solution

The proposed solution is this project where we aim to build an all in one platform suitable for GPGPU experimentation, learning and prototyping. We will develop a JavaScript runtime which aims to provide a bulk of features out the box to reduce the learning curve required and provide native bindings to popular industry standard APIs that are suitable to both novices and experts. The platform should be easy to install and easy to use, skipping lengthy and numerous SDK installations in favour of a onetime install platform that provides everything required out of the box. The development of the platform and research should highlight several key points. The first showing the speed of compilation and execution of JavaScript and how it can be utilized as a generic scripting language for numerous environments. The second showing how leveraging specialised hardware which is more common than ever in today's world can accelerate traditional applications. Finally, by showing the importance and relevance of both modern JavaScript as a general scripting language and GPGPU for being the tool that programmers must leverage if we are to see more performant software.

2.0 Technical Review

2.1 GPUs

The term Graphics Processing Unit (GPUs) was coined by NVidia when they released their graphics chip called the GeForce 256 (Nvidiacom, 2016). The origin of the modern GPU started in the 1970s where arcade manufactures to cut costs built systems with custom video chips to power the display. Today GPUs are an abundance, they are present in most computers including consoles, desktops, laptops, tablets, and mobile phones albeit in different forms. In the following section I will be providing a review of this key technology.

2.1.1 Hardware

On a hardware level GPUs are distinct. A Central Processing Unit (CPU) typically consists of a couple of cores, the most common being dual core processors and quad core processors. This contrasts with GPUs which commonly have more than 100 cores making them great at processing parallel workloads (Nvidiacom, 2009). GPUs are also distinct when it comes to memory. A traditional CPU will have multiple layers of cache in which to store data it will process. The cache is traditionally very small whereas GPUs have dedicated memory which was designed specifically for the GPU and often has a higher bandwidth than traditional system RAM. The ability for GPUs to accelerate computation workloads has now expanded the hardware to not only be used for 3D rendering but also for scientific research, data analysis, financial modelling, image processing and gas exploration.

2.1.2 Integrated vs Dedicated

GPUs to become more mainstream have been shrunk and extruded into different form factors to suit the computers they would be integrated into. Dedicated graphics cards are found in high end desktops, laptops, and workstations. They are installed into these computers via an expansion slot and are often the most powerful and expensive cards as they do not need to meet harsh size restraints or power limits. Traditionally integrated graphics were chips installed on the motherboard, however in 2010 Intel integrated the graphics chip onto the CPU die setting the stage for modern integrated graphics (Intelcom, 2016). The result was better media performance by default for standard CPUs as there was an increased demand for CPUs to be capable of moderate graphics tasks such as HD media playback and light 3D rendering. Intel was not the only CPU manufacture to do this. AMD also pioneered the technology with their Accelerated Processing Unit (APU) technology in 2011 which was designed to provide better 3D and media performance in small form factor computers such as laptops and game consoles.

2.1.2 NVidia & AMD

Two major chip manufacturers AMD and NVidia dominate the dedicated graphics market. There is a consensus that NVidia today holds a majority share of the market, this is backed up both by Steam hardware reports (Steampoweredcom, 2016) and research undertaken at John Peddie Research. Although NVidia dominates the market AMD is still an influential player. The latest generation consoles, the Xbox One and PlayStation 4 are powered by AMD graphics cards. What's more their Mantle API (Amdcom, 2016) was the starting point for the new API for both compute and graphics Vulkan (Khronosorg, 2016) which aims to supersede OpenGL and OpenCL entirely. AMD entered the graphics card market with the acquisition of ATI in 2006 and has been a keen player ever since. NVidia has its own accomplishments with its own compute API supported on its cards known as CUDA which is a direct competitor to OpenCL. NVidia cards are commonly the graphics card vendor of choice when it comes to laptops and general desktops as well.

2.1.3 Intel & AMD

Modern integrated graphics are now integrated onto the CPU die, making this technology completely dominated by the two major CPU manufacturers Intel and AMD. Intel added integrated graphics into their CPUs in 2010 with the launch of their Westmere microarchitecture. AMD arrived later with APUs based on their K10 architecture, that while not the first provided much better performance out of the box.

2.1.5 Software

Because GPUs are specialised hardware they have been traditionally accessed through industry approved API standards like OpenGL and OpenCL. Over the years, the number of APIs available have expanded as GPUs have evolved. The newest APIs include Vulcan (Khronosorg, 2016), Metal (Applecom, 2016) and CUDA (Nvidiacom, 2016). GPUs are traditionally used for parallel computation and advanced 3D rendering and in the following section I will be summarising the technology used most today to accomplish rendering and computation.

2.1.6 OpenGL & DirectX

3D Rendering has been traditionally accomplished through either DirectX or OpenGL. DirectX is a set of Windows APIs for multimedia applications. DirectX's key component is Direct3D which is a direct competitor to OpenGL and allows developers to write 3D applications. DirectX unlike OpenGL isn't cross platform, you will only find it on Windows, one of its key faults. Another key differential is that DirectX isn't backwards compatible unlike its competitor. OpenGL stands for Open Graphics Library; it is a cross platform API for 3D rendering. Unlike DirectX, OpenGL is only concerned with rendering and isn't a set of APIs but rather one API for rendering only. OpenGL is backwards compatible and uniquely has multiple versions which has seen it expand onto other platforms. OpenGL ES brought the API to embedded systems, and WebGL brought hardware accelerated rendering to the Web. In any case as it is one of the few platform independent graphics APIs it is still used today, most notably on Linux and Mac.

The key element to DirectX and OpenGL are programs called shaders. To provide more control of the rendering both APIs have programmable sections the programmer can use to dictate how data is rendered on screen. In DirectX such shaders are written in a language called HLSL or High Level Shading Language. In OpenGL these shaders are written in a language called GLSL or OpenGL Shading Language. An example of a simple orthographic OpenGL shader can be found in Figure 1.

```
layout(location = 0) in vec3 positions;
layout(location = 1) in vec3 uvs;

uniform mat4 projection;
uniform mat4 model;

out vec2 uv_cords;

void main() {
    vec4 position2D = (projection * model * vec4(positions, 1.0)).xy;
    gl_Position = vec4(position2D, 0.0, 1.0);
    uv_cords = uvs.xy;
}
```

Figure 1: Typical 2D OpenGL shader

2.1.7 CUDA & OpenCL

On the compute side of GPU APIs, we have CUDA and OpenCL. Apple originally proposed the Open Compute Library known as OpenCL to allow developers to take advantage of GPUs on their platform. They submitted it to the Kronos Group and it soon became an industry standard. Where OpenCL is different from CUDA is the range of devices it works on. OpenCL can run on any heterogeneous system and is not bound to a single operating system like DirectX or hardware manufacturer like CUDA. CUDA on the other hand will only run on NVidia hardware. Research (Karimi, K, 2016) found CUDA to perform better than OpenCL, however CUDAs inability to work across hardware from different manufacturers is certainly its biggest downfall, however it reserves its strength as the best performing API in the market.

Like the above mentioned rendering APIs CUDA and OpenCL have programmable elements called kernels where the programmer can dictate how data is transformed. In CUDA such kernels are written in CUDA C which is raw C/C++ with extensions allowing one to execute code on the GPU. In OpenCL kernels are written in OpenCL C which like CUDA mirrors the C/C++ language and adds extensions to fit the device it will run on. In Figure 2 you can see an OpenCL C kernel which performs a simple vector addition.

```
__kernel void vector_add(__global const float* a, __global const float* b, __global float* result)
{
    int gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

Figure 2: OpenCL kernel which performs a vector additional

2.1.8 Advantages of GPUs

Research (Yang et al, 2008) took bread and butter computer vision algorithms and compared their performance when processed across a CPU and GPU. With a histogram, they saw a 44x speed up when computed on the GPU. When it came to edge detection they saw a 200x speed up. Additionally, research (Teodoro et al, 2009) found that optimising a histopathology application resulted in a speed factor increase of between 19x to 40x in their tests. In computationally expensive tasks we can see GPUs can provide unseen speed ups in expensive computations. We can also see how a workbench could be advantageous to experiment and test such optimisations.

2.2 JavaScript

We chose JavaScript as the language for the platform for various reasons. The first is its speed. JavaScript has benefited from a large amount of investment in compiler development with most browser vendors now opting for Just in Time (JIT) compilers over traditional interpreters for JavaScript execution. The result is a tenfold increase in JavaScript speed making the language more suitable for high performance applications. Second JavaScript is a very popular language, in their yearly survey StackOverflow found JavaScript to be the most popular technology (Stack overflow blog, 2016) by a large mile, so using it for the platform would be advantageous as the language is popular with many developers. Finally, JavaScript has had a new recent standard ECMAScript 2015 which has sought to remove previous issues and present JavaScript as a clear concise general purpose scripting language rather than a language for document object model manipulation in the browser.

2.2.1 Typed Array Architecture

Recent revisions of the JavaScript standard have added support for objects designed to make low level programming possible (Mozillaorg, 2016). I will summarise the most ground breaking set of objects known as TypedArray objects as it now allows JavaScript to work with binary data directly. Typed Arrays were added in the JavaScript standard ECMAScript 2015 as the language lacked any ability to work with low level data and the typed array specification was an answer to this issue. They allow JavaScript to have types that represent raw C data types such as char and float. I will now cover these objects.

2.2.2 ArrayBuffer

ArrayBuffer is the base type for every Typed Array object and it just represents a stream of binary data. Look at the Figure 3 we can take the *struct person* and represent it in memory in JavaScript with the following ArrayBuffer shown in Figure 4.

```
struct person {
    double height;
    char name[256];
    int age;
};

person p;
```

Figure 3: basic struct example

```
class Person extends ArrayBuffer {
    constructor() {
        super(268); // 268 bytes

        this.height = new Uint32Array(this, 0, 1);
        this.name = new Uint8Array(this, 4, 256);
        this.age = new Float32Array(this, 260, 1);
    }
}

let p = new Person();
```

Figure 4: Figure 3 struct represented in modern JavaScript

At this point the JavaScript example and the C++ example have access to the same set of data and the same number of bytes in memory. This is an important step forward in JavaScript as it allows us to allocate and control bytes which was a concept absent from JavaScript till this point.

2.2.3 Uint8Array, Uint32Array, Float32Array, Float64Array

Following the base type ArrayBuffer you can now also represent arrays of bytes with greater precision than before. JavaScript numbers are defined in the standard as 64-bit double precision numbers. This limits control but with TypedArrays you can now control a greater range of integral types. Consider the following C++ arrays shown in Figure 5.

```
unsigned char characters[10];
unsigned int positive[100];

double ratios[100];
```

Figure 5: C style arrays

Previously it was impossible to have variables in JavaScript that natively mimicked these due to JavaScript having one type for all types of numbers. But due to the addition of TypedArrays this is no longer the case as can be seen in Figure 6.

```
let characters = new Uint8Array(10);
let positive = new Uint32Array(10);
let ratios = new Float64Array(100);
```

Figure 6: Figure 5 arrays represented in JavaScript

In short, the addition of these types to JavaScript better enables the language to interact with low level data structures and binary data. As such when building the platform using these objects has been prioritised as it stops the need to convert JavaScript data types to the data types found in C/C++.

2.3 V8 JavaScript Compiler

2.3.1 Interpreter vs Compiler

In 2008 Google set the benchmark for JavaScript compilers. They created a new JavaScript JIT compiler, V8 from the ground up to dramatically improve JavaScript execution speed. Browsers at the time used JavaScript interpreters instead. Internally they built a benchmark called V8 bench and measured performance increases overtime. As you can see in Figure 7 each subsequent revision of Chrome which in turn has a new version of V8 saw massive gains in JavaScript performance. This started the JavaScript compiler competition which saw all major JavaScript implementers drop their interpreters in favour of a JIT compiler in the hope that faster JavaScript would lead to a faster browser and better web experience. The key difference between an interpreter and a compiler is how the program is built and executed. Where an interpreter would typically execute one statement at a time, a compiler would translate the entire program into machine code ready to execute.

2.3.2 ECMAScript 2015

V8 implements ECMAScript as specified in ECMA-262, 5th edition, commonly referred as ECMAScript 2015 and runs on Windows, Mac OS X, and Linux systems. V8 enables any C++ application to expose its own objects and functions to JavaScript code. It's up to the developer to decide on the objects and functions exposed to JavaScript. There are many applications that use V8 already including Adobe Flash, the Dashboard Widgets in Apple's Mac OS X and Yahoo Widgets.

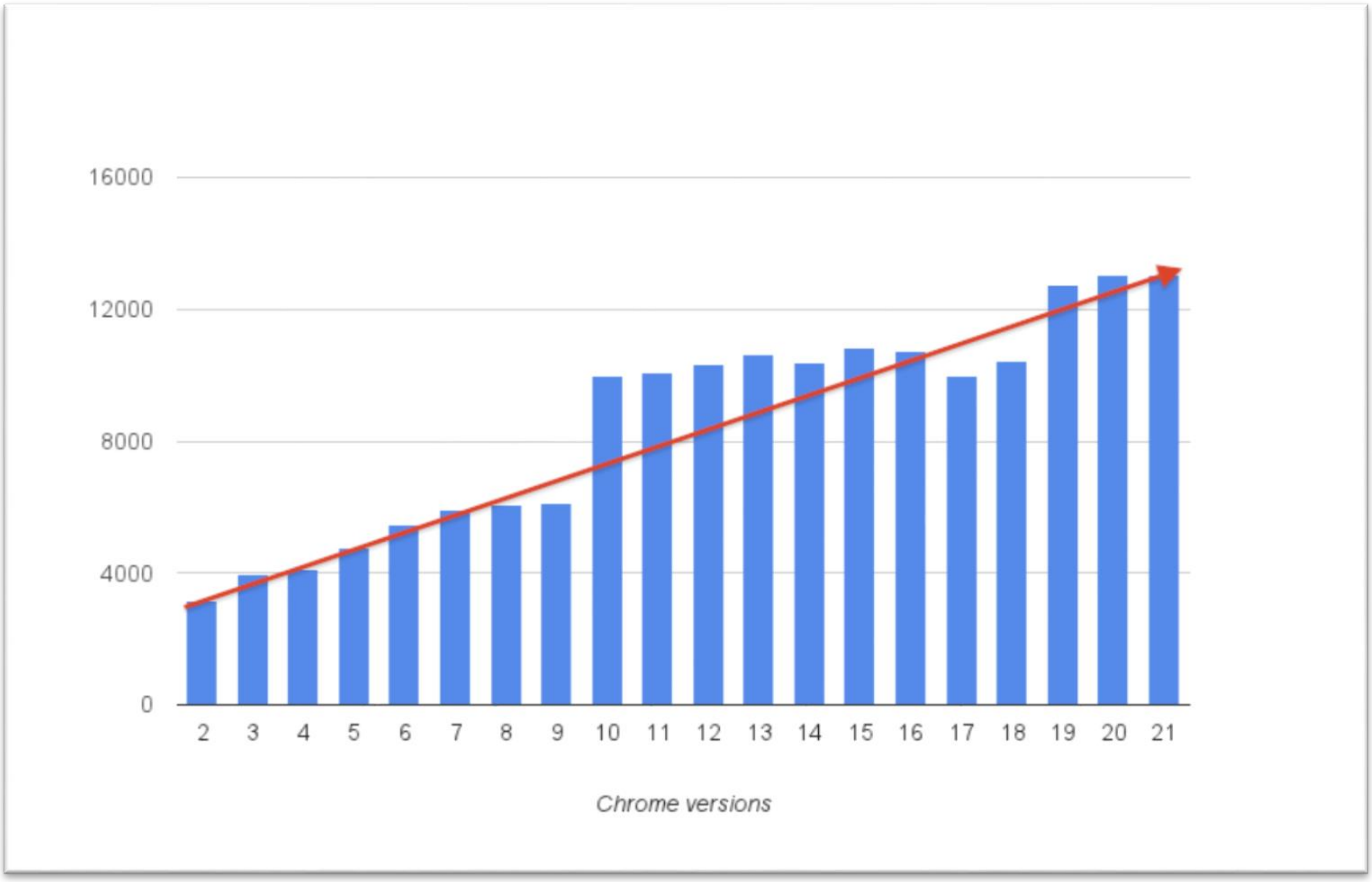


Figure 7: JavaScript benchmark scores with each release of the Chrome web browser.

2.3.3 Runtimes

Fast JavaScript execution did not go unnoticed. JavaScript can now be found in many environments other than the Web and in the programs written above. You can now write server side applications in JavaScript with Node.js (Nodejs foundation, 2016) which uses V8. You can write full 3D games with the Unity game engine (Unity3dcom, 2016) which uses it as its scripting language. Finally, through open source projects such as Electron (Atomio, 2016) you can now write native desktop applications as well. We built our platform on top of the V8 compiler to ensure that the platform is fast and efficient and provides access to the latest JavaScript standard and because it has been used so successfully in other runtimes.

3.0 Current Progress & Plan for Completion

3.1 Development Outline

The project is to develop a platform that allows GPU centric applications to be written in JavaScript to allow developers to experiment and develop hardware accelerated applications in a dynamically typed and flexible language. The development work consists of writing a C++ application which embeds Google's V8 JavaScript engine to compile and make available C++ functions to JavaScript. These functions would then direct calls to OpenCL and OpenGL for GPU code and common libraries for utilities such as window support, image loading and system information. In the next section I will be covering the steps that were taken to build our current prototype build of this C++ application which will serve as our platform.

3.2 Development Steps

3.2.1 Embedding V8

The first step was to get V8 downloaded and linked inside our C++ application. That was surprisingly difficult as V8 is not a small source project. As it was such a big project it had a lot of custom build tools and technologies that were needed to build V8 from source. V8's source can be built in multiple different ways, either with the GN meta build system or GYP meta build system. V8's repository is also built on top of Google's depot tools which must be installed as well and most of these technologies are poorly documented. After a large amount of time had been spent we managed to output V8 as a static library file which could now be linked to in a C++ application. Once there we followed the embedders guide which explains key concepts in V8.

3.2.1.1 Isolate

An isolate in V8 is defined as a virtual machine (VM) instance with its own heap. The idea is that an application should be able to spin up multiple VM instances from within a single application. You create an isolate like so using the C++ V8 API. This is the first object we create in our runtime to launch V8 and prepare for JavaScript execution. In Figure 8 you can see this object being created.

```
v8::Isolate* createIsolate(v8::Isolate::CreateParams& params)
{
   return v8::Isolate::New(params);
}
```

Figure 8: Sample function which creates a new VM instance

3.2.1.2 Handles

Handles are pointers to objects exposed to JavaScript. All V8 objects are accessed using handles and are needed as JavaScript uses a garbage collector and objects cannot be released until all handles are released. Handles come in many different varieties the most common one being *Local* which is just a stack allocated handle to the value stored in V8. In Figure 9 you can see a handle being created.

```
void createHandles(v8::Isolate* isolate)
{
    // Create new handle
    v8::Local<v8::Number> number = v8::Number::New(isolate, 10.0);
}
```

Figure 9: Sample function which creates a handle to a Number variable available in JavaScript

3.2.1.3 Scopes

Scopes are containers for a sequence of handles. They allow handles to be released on a function by function basis rather than by the primary scope. In Figure 10 all handles allocated in the current scope will be deleted when the *HandleScope* is deleted. Note to construct a *HandleScope* object you must pass the VM instance that the *HandleScope* will be run on. The *GetCurrent* function returns the current isolate.

```
void createScope()
{
    // All Handles declared in this function are deleted when this is destroyed.
    v8::HandleScope scope(v8::Isolate::GetCurrent());
}
```

Figure 10: Example of creating a handle scope

3.2.1.4 Context

A context is an execution environment that allows separate unrelated JavaScript code to run in a single instance of V8. Whenever you start up a V8 execution environment you must specify the context in which it runs. The contexts are used so you can have multiple JavaScript apps running at the same time, this is used to great effect in Chrome, where tabs have their own JavaScript context. Creating a context can be seen in Figure 11.

```
void createContext(v8::Isolate* isolate)
{
    // Create the global obejct for the context
    auto global = v8::ObjectTemplate::New(isolate);

    // Create our JavaScript context which can execute code
    auto context = v8::Context::New(isolate, nullptr, global);
}
```

Figure 11: Example of creating a context and global object template

3.2.2 Module System

Once V8 was successfully embedded and the source code written in a file was parsed and executed I looked at implementing a module system that would allow users to write modular code when using the runtime. I solved this by implementing the CommonJS standard which is used in the Node runtime as well. The CommonJS standard (Commonjsorg, 2016) specifies a contract for modules and how they should be handled.

3.2.2.1 Require

In the runtime, there should be a function called *require* which accepts a module identifier. The *require* function itself returns the exported contents of the foreign module. If, however the given module identifier does not lead to a valid module an error must be thrown with an acceptable message detailing why.

3.2.2.2 Module Context

In a module, which is normally a standalone JavaScript file there must be a variable called *require* which follows the above definition. There must also be a variable called *exports* which is an object that the module may add its API to as it executes. Finally, there must be a free variable *module* that is an object. This module object must have an id property and that module id value if passed to *require* should return itself.

3.1.2.3 Module Identifiers

A module identifier is a string delimited by forward slashes. If a module id has no filename extension ".js" is added by default. The module identifier is relative if it starts with ".". Finally, relative identifiers are resolved relative to the call to *require*.

3.2.3 Libraries

Once we had V8 embedded and a CommonJS module system implemented we wrote some basic libraries or common libraries for common tasks.

3.2.3.1 Console module

We provided a console module allowing users to write information to a console and read input from it as well. This is based on the console object found in most browsers for familiarity (Mozillaorg, 2016). A basic example of this module's functionality can be seen in Figure 12.

```
const console = require('console');
const input = console.read('Enter input:');

console.error('Error Log');
console.warn('Warn Log');
console.log('Log Log');
console.dir('Dir Log');

console.log(`You inputed ${input}`);
console.log(null, undefined);
console.log(10.0, -100, {});
```

Figure 12: Console API example

3.2.3.2 Datetime module

We also provided a date time module for managing time. These methods are based on the time browser specification so it's familiar to web developers (W3org, 2016). We also added an additional pause method which mirrors the Win32 API Sleep function. Figure 13 shows off some of the functions found in this module.

```
const { setTimeout, setInterval, clearInterval, pause } = require('datetime');
const console = require('console');

pause(100);

const number = setInterval(() => {
    for(let i = 0; i < 3; ++i) {
        console.log(`HelloWorld ${i}`)
    }
}, 1000);


setTimeout(() => clearInterval(number), 2500);
```

Figure 13: Datetime API example

3.2.3.3 System module

To provide information on the system we provided a system module. While we don't envisage this being part of an application we feel that a platform should provide useful information and this does that providing access to OS information, battery details, instruction sets and hardware information. Figure 14 shows the information available from this module.

```
const console = require('console');
const system = require('system');
const toString = str => JSON.stringify(str, null, 4)

console.log(toString(system.instructions));
console.log(toString(system.os));
console.log(toString(system.battery()));
console.log(toString(system.hardware))
```

Figure 14: System API example

3.2.3.4 Http module

JavaScript and JSON are prolific when it comes to services and data online. So, to provide access to content online, for instance JSON files we added a http module that allows the user to get content online which can then be streamed directly into an application. An example of this can be found in Figure 15.

```
const console = require('console');
const http = require('http');
const page = '/';
const port = 3000;

http.post('localhost', page, port, {}, (res, err) => console.log(res, err));
http.get('localhost', page, port, (res, err) => console.log(res, err));
```

Figure 15: Http API example

3.2.3.5 Fs module

Of course, a big feature needed for OpenCL and OpenGL is reading data off disk so we added a file system module which provides the ability to read text files, JSON files and images. Once read these objects can be passed directly to OpenCL and OpenGL for processing. Figure 16 gives a basic usage example.

```
const console = require('console');
const fs = require('fs');

console.log(fs.readJson('./vscode/launch.json'));
```

Figure 16: Fs API example

3.2.3.6 Display module

A key component of any OpenGL demo is the ability to render your graphics to a window. The display module was built as the one stop shop to handle windows, message boxes, and basic components available on desktop operating systems. In Figure 17 you can see an example of how to open a window and enable an OpenGL context.

```
const { openWindow } = require('display');
const console = require('console');

openWindow({}, window => {
    window.show();
    window.enableOpenGL();
    window.onFrame(() => {
        window.swapBuffers();
    });
});
```

Figure 17: Display API example

3.2.3.7 CL module & GL module

The core modules are the CL module and GL module which house the bindings to OpenCL and OpenGL. If you want to see these in action you can find them in the source code for the demos (see Appendix 2 and 3). The bindings written aim to mirror the APIs as much as possible by using concepts covered such as Typed Arrays for dealing with data buffers. If you look in Figure 18 you will see that by using the *with* keyword which takes all data in an object and makes it available outside (see Figure 19 for a better example) we have API calls that match as if it was in C++ and this is by design to make sure the code written maps as directly as possible to people with previous experience.

```
const console = require('console');
const gl = require('gl');

with (gl) {
    // Invalid Operation No GL Context
    console.log(glGetError() == 1282);
}
```

Figure 18: CL/GL API example

```
const point = {
    x: 100,
    y: 200
};

with (point) {
    console.log('X: ' + x); // x, y are now available
    console.log('Y: ' + Y);
}
```

Figure 19: With keyword example

3.2.4 Demonstrations

In addition to our draft runtime we also created two demos which shows us the bindings to both OpenCL and OpenGL from JavaScript working in full motion. I will quickly cover them and show example outputs. The code for the demonstrations can be found in the appendices (see Appendix 2 and 3).

3.1.4.1 Image Manipulator Demo

This demonstration loads a local image then opens an OpenCL context and executes an OpenCL kernel that outputs a result. The first kernel executed performs a grayscale operation on the image and writes it to an output buffer. The second kernel performs a simple blur across a given image and writes this to an output buffer. Upon completion, these output buffers are written to disk. Below you can find a sample output of the program. This demonstration is written entirely with the prototype runtime and certainly shows we are making fantastic progress. The output can be seen in Figure 20.

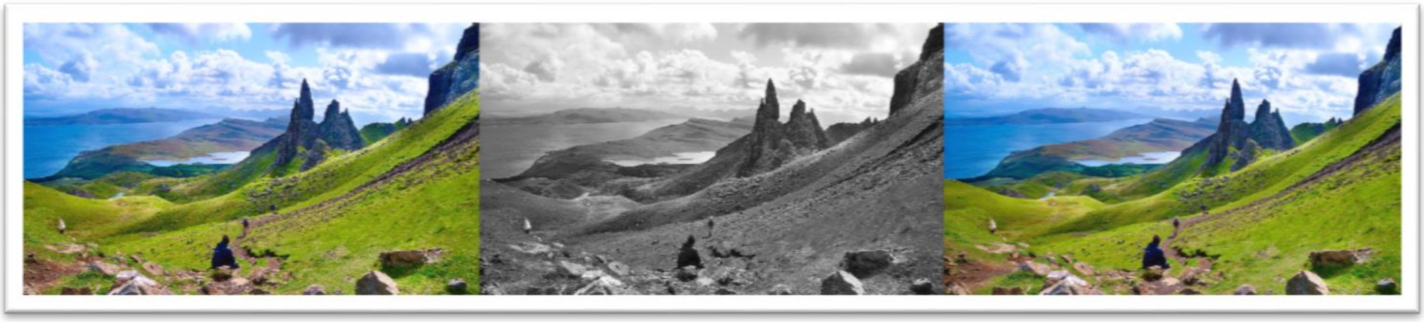


Figure 20: Original image followed by the grayscale output and blurred output

3.1.4.1 3D Textured Cube Demo

We also wrote this 3D textured cube demo which takes geometric data written in JSON and renders it on screen in 3D using OpenGL. The demo itself uses Vertex Array Objects (VAO) and Vertex Buffer Objects (VBO) to load and store vertex data on the GPU. It utilizes custom shaders and texture objects allowing us to bind textures to the geometry points specified. It also utilizes 3D matrices to perform translations, rotations and camera work to view the object in 3D. The output of the program can be seen in Figure 21.

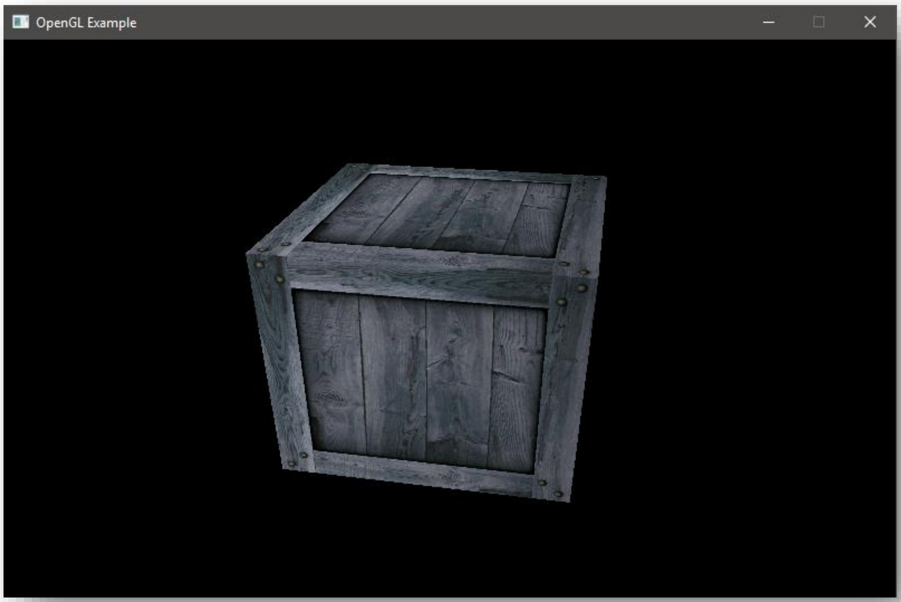


Figure 21: 3D rotating textured cube rendered using OpenGL

3.3 Development Technique

When developing the runtime, we followed an agile style when it came to development. In the following section I will be breaking down why I chose this style and how it affected the project's development.

3.3.1 Agile Software Development Methodology

Agile software development itself is a large umbrella term for a wide range of practices used to enforce the goals set out in their Agile Manifesto. The manifesto itself comprises of four values.

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contact negotiation
4. Responding to change over following a plan

3.3.2 Scrum Method

There are many methods that adopt an agile approach, I chose scrum as it's widely adopted and one I personally enjoy. Research has also been taken in this area to identify the advantages of scrum. Research (Mahalakshmi and Sundararajan, 2013) identified various advantages to using scrum including;

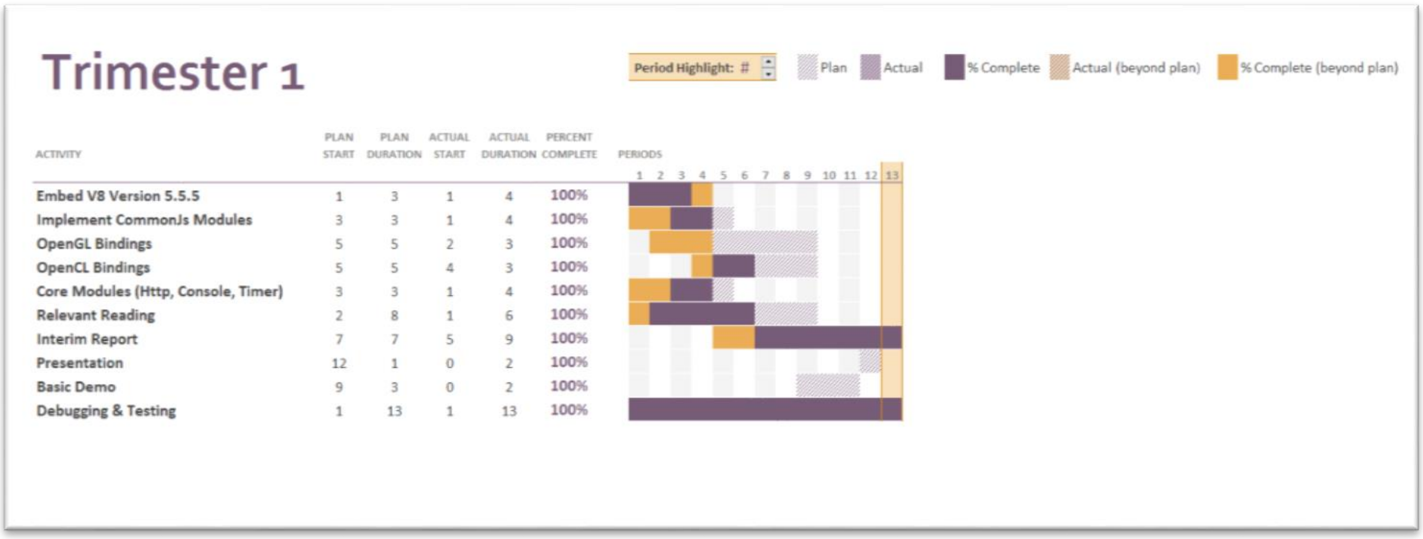
- Increased control of the project schedule
- Increased software quality
- Flexible enough when adapting to changes
- Work proceeds and completes more logically

3.3.3 Fortnightly Sprints

For each task, I managed a fortnightly sprint to achieve my work. This has worked wonders in the first part of the development and I will continue to use this for the final section of the project. When it came to assigning tasks into weekly sprints I would take one task from a Gantt chart, Figure 22 and break it down into multiple sub tasks which could be completed on a weekly or fortnightly basis.

3.3.4 Gantt Chart Schedule

As mentioned above to better manage the workload I created a Gantt chart to manage tasks and set out the schedule for the project. Note you can also find in the appendices copies of all the meetings I had with my supervisor which also helped coordinate progress and work load (see Appendix 4-7). Note that Figure 22 only covers the work done in trimester one.



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Embed V8 Version 5.5.5	1	3	1	4	100%
Implement CommonJs Modules	3	3	1	4	100%
OpenGL Bindings	5	5	2	3	100%
OpenCL Bindings	5	5	4	3	100%
Core Modules (Http, Console, Timer)	3	3	1	4	100%
Relevant Reading	2	8	1	6	100%
Interim Report	7	7	5	9	100%
Presentation	12	1	0	2	100%
Basic Demo	9	3	0	2	100%
Debugging & Testing	1	13	1	13	100%

Figure 22: Gantt chart with all major developments set out.

3.4 Plans for Completion

In the final parts of this report I will be covering the action points I will be focusing on in the incoming trimester to finish the progress made. Great progress has been achieved and we need to make sure it carries through to the next trimester.

3.4.1 Complete Bindings

Most importantly I will be looking to add most of the bindings required to OpenGL and OpenCL to build more advanced demonstrations to showcase the project. This may result in additional issues however, because both APIs have sections we have not touched upon which could prove difficult to expose to JavaScript without careful consideration. However, as we have seen OpenGL ES 2.0 made available to JavaScript through WebGL beforehand we don’t expect major issues here. So, the chances of a bottleneck here are small.

3.4.2 Professional Demonstrations

Once we have fully built out the bindings to OpenGL and OpenCL we will be looking to more advanced demos that prove the platform serves its purpose. Ideally, we will be looking at traditional methods that could help demonstrate the speeds up possible and how using the runtime can help prototype this work. Some early ideas include a GPU demo which solves a given path in a 3D level and a generic file renderer which takes render data specified in a JSON file and renders it in real time.

3.4.3 Presentation

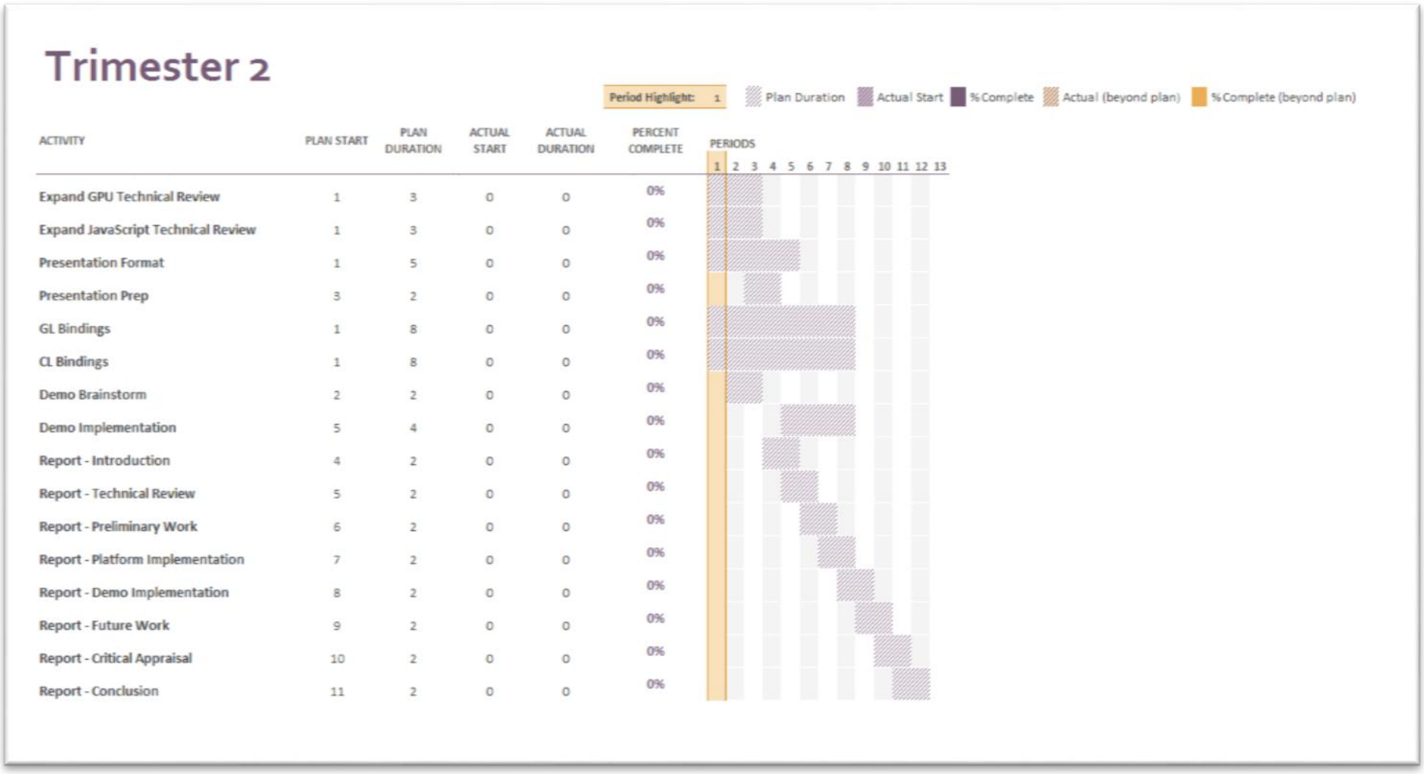
The presentation is due soon and a lot of preparation will need to be done. Luckily due to a good level of work being completed already I have no shortage of things I should demonstrate. I plan to do a proper presentation on the work done providing many opportunities to show the runtime in action through developed examples and demonstrations. If ready however we can use the more advanced demonstrations that we will construct in trimester two.

3.4.4 Final Honours Project Report

Finally, the most important piece that needs to be completed is the final honours project report which is the bulk of the marks for the project. I plan to make an early start on this to ensure this isn't rushed in the last weeks of trimester two. The technical review from this report will certainly be expanded on when it comes to the honours report but the good progress here will give me a good starting point.

3.4.5 Trimester 2 Gantt Chart

To prepare for the future trimester I took all the above points mentioned and merged them into a friendly Gantt chart which will help me measure my progress and keep me on track. The Gantt chart is visible in Figure 23 and covers tasks only for trimester two.



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Expand GPU Technical Review	1	3	0	0	0%
Expand JavaScript Technical Review	1	3	0	0	0%
Presentation Format	1	5	0	0	0%
Presentation Prep	3	2	0	0	0%
GL Bindings	1	8	0	0	0%
CL Bindings	1	8	0	0	0%
Demo Brainstorm	2	2	0	0	0%
Demo Implementation	5	4	0	0	0%
Report - Introduction	4	2	0	0	0%
Report - Technical Review	5	2	0	0	0%
Report - Preliminary Work	6	2	0	0	0%
Report - Platform Implementation	7	2	0	0	0%
Report - Demo Implementation	8	2	0	0	0%
Report - Future Work	9	2	0	0	0%
Report - Critical Appraisal	10	2	0	0	0%
Report - Conclusion	11	2	0	0	0%

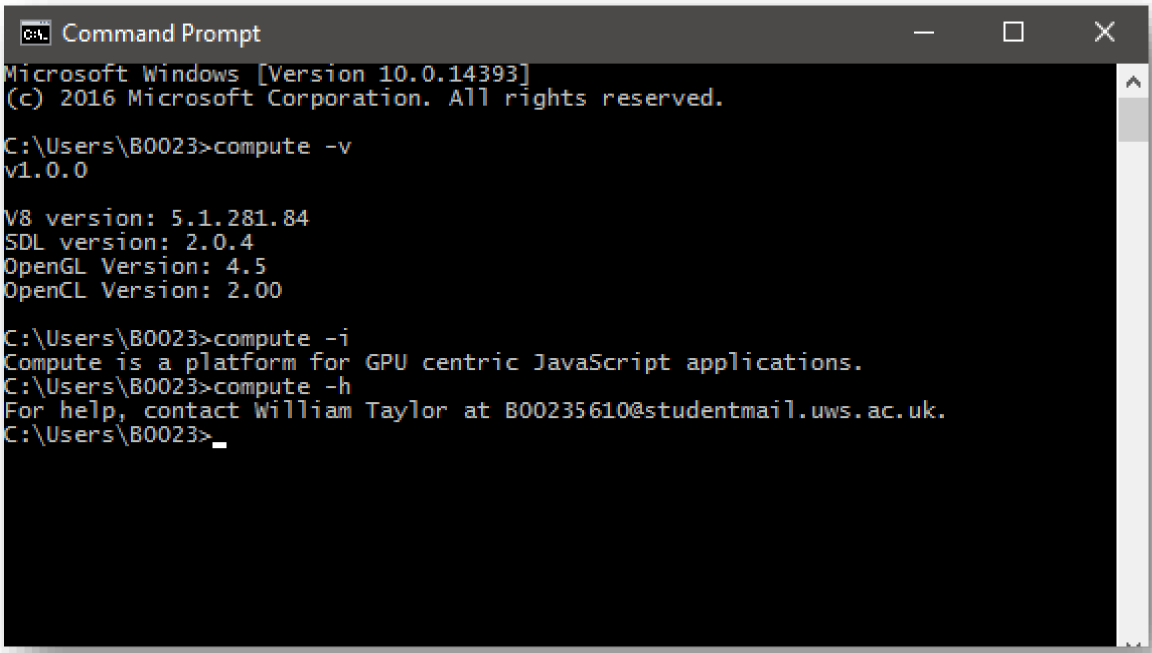
Figure 23: Gantt chart with the tasks that need to be completed in trimester two.

4.0 Concluding Remarks

4.1 Excellent Progress

Progress has been extremely good in my view. Not only do we have a working version of the runtime which is capable of compiling modern JavaScript and providing industry standard bindings to GPU APIs but working demonstrations as well. Most implementation work for the honours project is handled in trimester two and to make such progress will certainly help me in the incoming trimester. This runtime isn't just an unstable prototype either it is packed with loads of features including:

- Basic OpenCL & OpenGL bindings
- Common Modules
- Fast JavaScript execution
- External Modules via NPM
- CLI utilities (See Figure 24)



```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\B0023>compute -v
v1.0.0

V8 version: 5.1.281.84
SDL version: 2.0.4
OpenGL Version: 4.5
OpenCL Version: 2.00

C:\Users\B0023>compute -i
Compute is a platform for GPU centric JavaScript applications.
C:\Users\B0023>compute -h
For help, contact William Taylor at B00235610@studentmail.uws.ac.uk.
C:\Users\B0023>_
```

Figure 24: Providing argument -v shows OpenCL + OpenGL versions on the system

4.2 Possible Future Problems

We could have problems in the future as we seek to provide more bindings to industry APIs. As these APIs have been built up over years with many new additions and extensions it may prove difficult to provide a complete set of bindings that are bug free due to the amount of functions available in each API. Nevertheless, we will seek to provide proper, tested, working calls to the most popular methods in these APIs so we can build the demonstrations required to show that prototyping applications using this platform is advantageous and reliable.

4.3 Conclusion

In conclusion, a lot of work has been completed and we are well on track to producing a professional piece of research. The following months will bring forward much advancement but all were made possible thanks to the addition of early prototype work and building on this will be the key to a good result at the end of the module.

I am most impressed with how flexible JavaScript has been when integrating the language with the OpenCL and OpenGL APIs. It makes one wonder of the direction of travel for GPGPU. I have seen first-hand the advantages of making access to this hardware easier through higher level languages and how by presenting an all in one package how simple a demo could be to make in such little time. It certainly makes things easier and allows more time to focus on what you want to do.

I have also been shocked by how much can be done with V8 and JavaScript, looking at the demos it’s hard to believe that the application is raw JavaScript and that by modifying a single script file I can completely change the output. Consider if these demos were a single C++ source file. JavaScript has quite successfully come in and replaced C++ while still providing the same control and is still relatively fast and that in my view is amazing for a language that was once considered so limited. As we move forward it will be interesting to see more advantages reveal themselves as this flexible runtime for learners and developers makes programming for GPUs as fun and easy as it should be.

References

Amdcom. 2016. *Amdcom.* [Online]. [18 December 2016]. Available from: http://www.amd.com/en-us/innovations/software-technologies/mantle

Amdcom. 2016. *Amdcom.* [Online]. [5 December 2016]. Available from: http://www.amd.com/en-gb/products/processors/desktop/a-series-apu

Applecom. 2016. *Applecom.* [Online]. [18 December 2016]. Available from: https://developer.apple.com/metal/

Atomio. 2016. *Electron.* [Online]. [13 December 2016]. Available from: http://electron.atom.io/

Commonjsorg. 2016. *Commonjs.org.* [Online]. [5 December 2016]. Available from: http://wiki.commonjs.org/wiki/Modules/1.1.1

Ecma International. (2015). *ECMAScript Specification.* Available from: http://www.ecma-international.org/ecma-262/6.0/

David Geer, 2005. Chip makers turn to multicore processors. *Computer*, *38*(5), pp.11-13

Googlesourcecom. 2016. Googlesource.com. [Online]. [5 December 2016]. Available from: https://chromium.googlesource.com/v8/v8.git

Intelcom. 2016. *Intel® ARK (Product Specs).* [Online]. [18 December 2016]. Available from: http://ark.intel.com/products/43546

Intelcom. 2016. *Intel.* [Online]. [5 December 2016]. Available from: http://www.intel.com/content/www/us/en/architecture-and-technology/visual-technology/graphics-overview.html

Khronosorg. 2016. *Khronosorg.* [Online]. [18 December 2016]. Available from: https://www.khronos.org/vulkan/

Karimi, K., Dickson, N.G. and Hamze, F., 2010. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*.

Mahalakshmi, M. and Sundararajan, M., 2013. Traditional SDLC Vs Scrum Methodology–A Comparative Study. *International Journal of Emerging Technology and Advanced Engineering*, *3*(6), pp.192-196.

Nodejs foundation. 2016. *Nodejsorg.* [Online]. [18 December 2016]. Available from: https://nodejs.org/en/

Mozillaorg. 2016. *Mozilla Developer Network.* [Online]. [6 December 2016]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

Mozillaorg. 2016. *Mozilla Developer Network.* [Online]. [5 December 2016]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Console

Nvidiacom. 2016. *Nvidiacom.* [Online]. [15 December 2016]. Available from: http://www.nvidia.com/object/IO_20020111_5424.html

Nvidiacom. 2009. *The Official NVIDIA Blog.* [Online]. [15 December 2016]. Available from: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

Nvidiacom. 2016. *Nvidiacom.* [Online]. [18 December 2016]. Available from: http://www.nvidia.com/object/cuda_home_new.html

Steampoweredcom. 2016. *Steampoweredcom.* [Online]. [18 December 2016]. Available from: http://store.steampowered.com/hwsurvey/

Shen, G., Gao, G.P., Li, S., Shum, H.Y. and Zhang, Y.Q., 2005. Accelerate video decoding with generic GPU. *IEEE Transactions on circuits and systems for video technology*, *15*(5), pp.685-693.

Stack overflow blog. 2016. *Stack Overflow Blog.* [Online]. [18 December 2016]. Available from: https://stackoverflow.blog/2016/03/stack-overflow-developer-survey-results/

Unity3dcom. 2016. *Unity.* [Online]. [18 December 2016]. Available from: https://unity3d.com/

W3org. 2016. *W3org.* [Online]. [5 December 2016]. Available from: https://www.w3.org/TR/2011/WD-html5-20110525/timers.html

Yang, Z., Zhu, Y. and Pu, Y., 2008, December. Parallel image processing based on CUDA. In *Computer Science and Software Engineering, 2008 International Conference on* (Vol. 3, pp. 198-201). IEEE.

Appendix 1 – Project Specification

COMPUTING HONOURS PROJECT SPECIFICATION FORM

Project Title: A JavaScript Runtime for Hardware Accelerated Applications.

Student: William Taylor **Banner ID:** B00235610

Supervisor: Paul Keir

Moderator: Mark Stansfield

Outline of Project:
The research is to develop a platform that allows GPU centric applications to be written in JavaScript. The platform's goal is to provide compete bindings to industry standard GPU libraries (OpenCL & OpenGL) to allow developers to experiment and develop hardware accelerated applications in a dynamically typed and flexible language. The platform aims to expand the JavaScript ecosystem of runtimes and provide a workbench for those keen on the performance gains hardware acceleration can bring.

The research will highlight a number of key points. The first showing the speed of compilation and execution of JavaScript. The second showing how leveraging specialised hardware can accelerate traditional applications. Finally, the importance of accelerated programming and JavaScript to the technology sector.

A Passable Project will:

- *Showcase a generalised GPU demonstration written in JavaScript.*
- *Will do an analysis of current GPU technologies in JavaScript and how they can be leveraged.*

A First Class Project will:

- *Develop and make available a platform that allows JavaScript developers to write generalised hardware accelerated applications.*
- *Showcase several generalised GPU demonstrations written in JavaScript.*
- *Do in depth research into future and current JavaScript technology which enables hardware acceleration.*
- *Research and demonstrate the advantages of JavaScript over other dynamic languages (e.g. Python) in developing hardware accelerated applications.*

Reading List:
1. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.3*
2. *Programming 3D applications with HTML5 and WebGL*
3. *Heterogeneous computing with OpenCL*
4. *OpenCL Programming Guide*

Resources Required:
Visual Studio, OpenCL & OpenGL enabled hardware, Chrome's V8 JavaScript JIT compiler, Git + Github.

Marking Scheme:	**Marks**
Introduction	10
Area Overview	15
Requirements and Design	10

Development	30
Project Demonstrations	20
Critical Self-Appraisal	5
Conclusions and Recommendations	10

Signed:

Student	**Supervisor**	**Moderator**	**Year Leader**

IMPORTANT: ***By signing this form all signatories are confirming that any potential ethical issues have been considered and necessary actions undertaken and that Mark Stansfield (Module Coordinator) and Malcolm Crowe (Chair of School Ethics Committee) have been informed of any potential ethical issues relating to this proposed Hons Project.***

Appendix 2 – OpenGL Demo Source Code

```
const { openWindow } = require('display');
const console = require('console');
const glm = require('gl-matrix');
const gl = require('gl');
const fs = require('fs');

const shaders = { vs: fs.read('./vert.glsl'), fs: fs.read('./frag.glsl') };
const geometry = fs.readJson('./cube.json');
const texture = fs.readImage('./crate.jpg');

function createShader(shaderType, shaderSource) {
    with (gl) {
        const shader = glCreateShader(shaderType);
        glShaderSource(shader, shaderSource.length, shaderSource, 0);
        glCompileShader(shader);

        const compileStatus = [,], logLength = [,];
        glGetShaderiv(shader, GL_COMPILE_STATUS, compileStatus);
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, logLength);

        if (compileStatus[0] != GL_TRUE) {
            const log = [,];
            glGetShaderInfoLog(shader, logLength[0], logLength[0], log)
            console.log('Compile error', log[0]);
        }

        return shader;
    }
}

openWindow({ w: 800, h: 500 }, window => {
    window.setTitle('OpenGL Example');
    window.show();
    window.enableOpenGL();

    with (gl) {
        const vs = createShader(GL_VERTEX_SHADER, shaders.vs.contents);
        const fs = createShader(GL_FRAGMENT_SHADER, shaders.fs.contents);
        const program = glCreateProgram();

        glAttachShader(program, vs);
        glAttachShader(program, fs);
        glLinkProgram(program);
        glUseProgram(program);

        glEnable(GL_DEPTH_TEST);
        glEnable(GL_TEXTURE_2D);

        const vertexArrayObject = new Uint32Array(1);
        const vertexBuffer = new Uint32Array(1);
        const colourBuffer = new Uint32Array(1);
        const uvBuffer = new Uint32Array(1);
        const textureID = new Uint32Array(1);

        glGenTextures(1, textureID);
        glBindTexture(GL_TEXTURE_2D, textureID[0]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texture.width, texture.height, 0, GL_RGBA, GL_UN-
SIGNED_BYTE, texture);
        glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        const vertexData = Float32Array.from(geometry.cube);
        const colourData = Float32Array.from(geometry.colour);
        const uvData = Float32Array.from(geometry.uvs);

        glGenVertexArray(1, vertexArrayObject);
        glBindVertexArray(vertexArrayObject[0]);

        glGenBuffers(1, vertexBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER, vertexData.byteLength, vertexData, GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

        glGenBuffers(1, colourBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, colourBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER, colourData.byteLength, colourData, GL_STATIC_DRAW);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

        glGenBuffers(1, uvBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, uvBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER, uvData.byteLength, uvData, GL_STATIC_DRAW);
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, 0);

        const projection = glm.mat4.create();
        const model = glm.mat4.create();
        const view = glm.mat4.create();

        glm.mat4.perspective(projection, glm.glMatrix.toRadian(45.0), 4.0 / 3.0, 0.1, 100.0);
        glm.mat4.lookAt(view,
            glm.vec3.fromValues(4, 3, -3),
            glm.vec3.fromValues(0, 0, 0),
            glm.vec3.fromValues(0, 1, 0)
        );

        window.onFrame(() => {
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glClearColor(0.0, 0.0, 0.0, 0.0);

            glm.mat4.rotate(model, model, glm.glMatrix.toRadian(1.0), glm.vec3.fromValues(0, 1, 0));

            const projectionLocation = glGetUniformLocation(program, "projection");
            const modelLocation = glGetUniformLocation(program, "model");
            const viewLocation = glGetUniformLocation(program, "view");

            glUniformMatrix4(projectionLocation, 1, GL_FALSE, Float32Array.from(projection));
            glUniformMatrix4(modelLocation, 1, GL_FALSE, Float32Array.from(model));
            glUniformMatrix4(viewLocation, 1, GL_FALSE, Float32Array.from(view));
            glDrawArrays(GL_TRIANGLES, 0, 36);

            window.swapBuffers();
        });

        window.onClose(() => {
            fs.freeImage(texture);
        });
    }
});
```

Appendix 3 – OpenCL Demo Source Code

```
const randomArray = require('random-array');
const console = require('console');
const http = require('http');
const cl = require('cl');
const fs = require('fs');

function acquirePlatform() {
    const platforms = [];

    with (cl) {
        clGetPlatformIDs(0, null, platforms);
        clGetPlatformIDs(platforms.length, platforms, null);

        platforms.forEach(platform => {
            clGetPlatformInfo(platform, CL_PLATFORM_NAME);
            clGetPlatformInfo(platform, CL_PLATFORM_VERSION);
            clGetPlatformInfo(platform, CL_PLATFORM_PROFILE);
            clGetPlatformInfo(platform, CL_PLATFORM_VENDOR);
        });
    }

    return platforms[0];
}

function acquireDevice(platform) {
    const devices = [];

    with (cl) {
        clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, null, devices);
        clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, devices.length, devices, null);

        devices.forEach(device => {
            clGetDeviceInfo(device, CL_DEVICE_NAME);
            clGetDeviceInfo(device, CL_DEVICE_VERSION);
            clGetDeviceInfo(device, CL_DEVICE_VENDOR);
        });
    }

    return devices[0];
}

function image(program, context, commandQueue, kernalName, inputName, outputName) {
    with (cl) {
        const kernel = clCreateKernel(program, kernalName, null);
        const img = fs.readImage(inputName);

        const format = {}, error = {};
        format.image_channel_order = CL_RGBA;
        format.image_channel_data_type = CL_UNORM_INT8;

        const imageMemory = [, , ];
        imageMemory[0] = clCreateImage2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, format,
img.width, img.height, 0, img.data, error);
        imageMemory[1] = clCreateImage2D(context, CL_MEM_READ_WRITE, format, img.width, img.height, 0,
null, error);
```

```
        const sampler = clCreateSampler(context, CL_FALSE, CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_NEAREST,
error);
        const globalWorkSize = Uint32Array.from([1920, 1200]);
        const localWorkSize = Uint32Array.from([16, 16]);

        clSetKernelArg(kernel, 0, 4, imageMemory[0]);
        clSetKernelArg(kernel, 1, 4, imageMemory[1]);
        clSetKernelArg(kernel, 2, 4, sampler);
        clSetKernelArg(kernel, 3, 4, img.width);
        clSetKernelArg(kernel, 4, 4, img.height);

        const output = new ArrayBuffer(img.width * img.height * 4);
        const region = Uint32Array.from([img.width, img.height, 1]);
        const origin = Uint32Array.from([0, 0, 0]);

        clEnqueueNDRangeKernel(commandQueue, kernel, 2, null, globalWorkSize, localWorkSize, 0, null,
null);
        clEnqueueReadImage(commandQueue, imageMemory[1], CL_TRUE, origin, region, 0, 0, output, 0, null,
null);

         fs.writeImage(outputName, output, img.width, img.height);
         fs.freeImage(img);

         clReleaseSampler(sampler);
         clReleaseKernel(kernel);
    }
}

const platform = acquirePlatform();
const device = acquireDevice(platform);
const source = fs.read('./kernel.cl');

with (cl) {
    const properties = [CL_CONTEXT_PLATFORM, platform, 0];
    const context = clCreateContextFromType(properties, CL_DEVICE_TYPE_ALL, null, null, null);
    const commandQueue = clCreateCommandQueue(context, device, 0, null);

    const program = clCreateProgramWithSource(context, 1, source.contents, null, null);
    const err = clBuildProgram(program, 0, null, null, null, null);

    if (err != CL_SUCCESS) {
        const buildLog = {};
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 1, buildLog, null);
        console.log(buildLog.log);
    }

    image(program, context, commandQueue, "gaussian_filter", "image.jpg", "blur.png");
    image(program, context, commandQueue, "grayscale", "image.jpg", "grayscale.png");

    clReleaseCommandQueue(commandQueue);
    clReleaseContext(context);
    clReleaseProgram(program);
}
```

Appendix 4 – Supervisor Meeting 1

PROGRESS AND MANAGEMENT MEETING AGENDA

Student: William Taylor

Supervisor: Paul Keir

Meeting Number: 1

Date/Time: 12:00 19/09/2016

PROGRESS

- Submitted draft project specification.
- Bought new books on OpenGL& OpenCL.
- Acquired new hardware for my home PC.
- Got a HelloWorld demo setup for my idea.
- Made improvements to my project specification after consultation.

AGENDA FOR FORMAL MEETING

1. Final review of project specification
2. Reflection on early work
3. Idea generator for impactful demonstrations
4. Review on scope and on likelihood of a successful project which stands out.
5. Date of next formal meeting

MANAGEMENT MEETING MINUTES AND PLAN

Student: William Taylor

Supervisor: Paul Keir

Meeting Number: 1

Date/Time: 12:00 19/09/2016

MINUTES

The following tasks and issues were discussed and specific actions agreed:

1. Summarised basic process
2. Discussed project ambitions
3. Discussed project potential
4. Demonstrated current work
5. Final review of project specification
6. Discussed project ideas

PLAN

For the next month:

- Submit project specification
- Have basic OpenCL & OpenGL functions bound
- Figure out demonstration ideas.
- Figure out how to demonstrate the project through media eg blogs.

Appendix 5 – Supervisor Meeting 2

PROGRESS AND MANAGEMENT MEETING AGENDA

Student: William Taylor **Supervisor:** Paul Keir

Meeting Number: 2 **Date/Time:** 12:00 26/09/2016

PROGRESS

- Refactored solution
- Finalised project specification
- Better window support.
- Background thread system
- Dealt with race conditions
- Tweaked marking scheme

AGENDA FOR FORMAL MEETING

1. Final review of project specification.
2. Summarise project goals.
3. General points of view for a good project.

MANAGEMENT MEETING MINUTES AND PLAN

Student: William Taylor **Supervisor:** Paul Keir

Meeting Number: 2 **Date/Time:** 12:00 26/09/2016

MINUTES

The following tasks and issues were discussed and specific actions agreed:

1. Ways to maximise marks with the project
2. Project specification signed off and reviewed.
3. Weekly meetings agreed with an informal format

PLAN

- Gantt chart made and uploaded to the GitHub repository
- Make more progress on OpenGL + OpenCL books
- Start interim report template
- Finish core features of the platform.

Appendix 6 – Supervisor Meeting 3

PROGRESS AND MANAGEMENT MEETING AGENDA

Student: William Taylor **Supervisor:** Paul Keir

Meeting Number: 3 **Date/Time:** 10:30 20/10/2016

PROGRESS

- Working on debugging
- Working version of external libs
- Literature review
- Presentation plan

AGENDA FOR FORMAL MEETING

1. Finalise design for the literature review
2. Finalise design for the presentation
3. Final review points for December

MANAGEMENT MEETING MINUTES AND PLAN

Student: William Taylor **Supervisor:** Paul Keir

Meeting Number: 3 **Date/Time:** 10:30 20/10/2016

MINUTES

The following tasks and issues were discussed and specific actions agreed:

1. Showed ability to load modules via npm
2. Discussed debug ability
3. Discussed the presentation plan
4. Discussed the plan for the interim report
5. Reviewed sources used for writing the interim report
6. Set out the tasks for next week

PLAN

The following tasks and timelines have been agreed both for the next month and beyond:

- More bindings for OpenCL + OpenGL
- Additional user testing to make sure bugs are at an all-time low
- Build basic template for the presentation
- Build basic template for the interim report

Appendix 7 – Supervisor Meeting 4

PROGRESS AND MANAGEMENT MEETING AGENDA

Student: William Taylor	**Supervisor:** Paul Keir
Meeting Number: 4	**Date/Time:** 12:00 31/10/2016

PROGRESS

- Failed on debugging for now
- Got new OpenGL bindings
- Shader support
- OpenCL C bindings now available
- Basic OpenCL kernel program written

AGENDA FOR FORMAL MEETING

1. Discuss current progress of binding APIs
2. Look at additional topics for the literature review
3. Progress review for the trimester
4. Possible change to project specification

MANAGEMENT MEETING MINUTES AND PLAN

Student: William Taylor	**Supervisor: Paul Keir**
Meeting Number: 4	**Date/Time:** 12:00 31/10/2016

MINUTES

The following tasks and issues were discussed and specific actions agreed:

1. Showed the new OpenGL demo
2. Showed the new OpenCL demo
3. Showed the usage of typed arrays
4. Talked about changing the specification
5. Talked about progress towards a good grade

PLAN

The following tasks and timelines have been agreed both for the next month and beyond.

- To flesh out the demos for the honours presentation and start work on the interim report