如何实现前端高性能计算？

最近做一个项目，里面涉及到在前端做大量计算，直接用js跑了一下，大概需要15s的时间，←也就是用户的浏览器会卡死15s，这个完全接受不了。

虽说有V8这样牛逼的引擎，但大家知道js并不适合做CPU密集型的计算，一是因为单线程，二是因为动态语言。我们就从这两个突破口入手，首先搞定“单线程”的限制，尝试用WebWorkers来加速计算。

前端高性能计算之一：WebWorkers

什么是WebWorkers

简单说，WebWorkers是一个HTML5的新API，web开发者可以通过此API在后台运行一个脚本而不阻塞UI，可以用来做需要大量计算的事情，充分利用CPU多核。

大家可以看看这篇文章介绍https://www.html5rocks.com/en/tutorials/workers/basics/，或者对应的中文版。

> The Web Workers specification defines an API for spawning background scripts in your web application. Web Workers allow you to do things like fire up long-running scripts to handle computationally intensive tasks, but without blocking the UI or other scripts to handle user interactions.

可以打开这个链接自己体验一下WebWorkers的加速效果

直接使用WebWorkers接口还是太繁琐，好在有人已经对此作了封装：Parallel.js。

注意Parallel.js可以通过node安装：

```
$ npm install paralleljs
```

不过这个是在node.js下用的，用的node的cluster模块。如果要在浏览器里使用的话，需要直接应用js:

```
<script src="parallel.js"></script>
```

然后可以得到一个全局变量，`Parallel`。`Parallel`提供了`map`和`reduce`两个函数式编程的接口，可以非常方便的进行并发操作。

我们先来定义一下我们的问题，由于业务比较复杂，我这里把问题简化成求1-1,0000,0000的和，然后在依次减去1-1,0000,0000，答案显而易见：0！这样做是因为数字太大的话会有数据精度的问题，两种方法的结果会有一些差异，会让人觉得并行的方法不可靠。此问题在我的mac pro chrome61下直接简单地跑js运行的话大概是1.5s（我们实际业务问题需要15s，这里为了避免用户测试的时候把浏览器搞死，我们简化了问题）。


```
const N = 100000000;// 总次数1亿

// 更新自2017-10-24 16：47：00
// 代码没有任何含义，纯粹是为了模拟一个耗时计算，直接用
//   for (let i = start; i <= end; i += 1) total += i;
// 有几个问题，一是代码太简单没有任何稍微复杂一点的操作，后面用C代码优化的时候会优化得很夸张，没法对比。
// 二是数据溢出问题，  我懒得处理这个问题，下面代码简单地先加起来，然后再减掉，答案显而易见为0，便于测试。
```

```
    }
  }

  return total;
}

function paraSum(N) {
  const N1 = N / 10;//我们分成10分，没分分别交给一个web worker，
parallel.js会根据电脑的CPU核数建立适量的workers
  let p = new Parallel([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    .require(sum);
  return p.map(n => sum((n - 1) * 10000000 + 1, n * 10000000))//
在parallel.js里面没法直接应用外部变量N1
    .reduce(data => {
      const acc = data[0];
      const e = data[1];
      return acc + e;
    });
}

export { N, sum, paraSum }
```

代码比较简单，我这里说几个刚用的时候遇到的坑。

require所有需要的函数

比如在上诉代码中用到了 `sum` ，你需要提前 `require(sum)` ，如果sum中由用到了另一个函数 `f` ，你还需要 `require(f)` ，同样如果 `f` 中用到了 `g` ，则还需要 `require(g)` ，直到你require了所有用到的定义的函数。。。。

没法 `require` 变量

我们上诉代码我本来定义了 `N1` ，但是没法用

ES6 编译成 ES5 之后的问题以及Chrome没报错

另外我后来在同样的电脑上Firefox55.0.3（64位）测试，上诉代码居然只要190ms！！！在Safari9.1.1下也是190ms左右。。。

Refers

- https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Workers_API/Using_web_workers
- https://www.html5rocks.com/en/tutorials/workers/basics/
- https://parallel.js.org/
- https://johnresig.com/blog/web-workers/
- http://javascript.ruanyifeng.com/htmlapi/webworker.html
- http://blog.teamtreehouse.com/using-web-workers-to-speed-up-your-javascript-applications

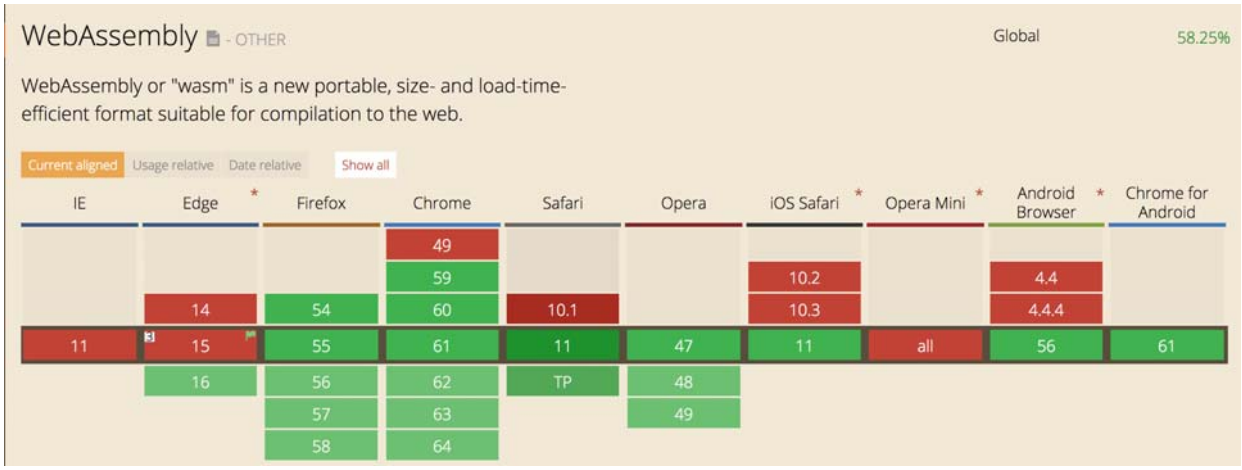
---

前端高性能计算之二：asm.js & webassembly

前面我们说了要解决高性能计算的两个方法，一个是并发用WebWorkers，另一个就是用更底层的静态语言。

2012年，Mozilla的工程师Alon Zakai在研究LLVM编译器时突发奇想：能不能把C/C++编译成Javascript，并且尽量达到Native代码的速度呢？于是他开发了Emscripten编译器，用于将C/C++代码编译成Javascript的一个子集asm.js，性能差不多是原生代码的50%。大家可以看看这个PPT。

之后Google开发了[Portable Native Client][PNaCl]，也是一种能让浏览器运行C/C++代码的技术。
后来估计大家都觉得各搞各的不行啊，居然Google, Microsoft, Mozilla, Apple等几家大公司一起合作开发了一个面向Web的通用二进制和文本格式的项目，那就是WebAssembly，官网上的介绍是：



安装Emscripten

访问https://kripken.github.io/emscripten-site/docs/getting_started/downloads.html

1. 下载对应平台版本的SDK

2. 通过emsdk获取最新版工具

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

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

# Make the "latest" SDK "active" for the current user. (writes
~/.emscripten file)
./emsdk activate latest

# Activate PATH and other environment variables in the current
terminal
source ./emsdk_env.sh
```

```
LLVM_ROOT = os.path.expanduser(os.getenv('LLVM', '/home/ubuntu/a-
path/emscripten-fastcomp/build/bin'))
```

5. 验证是否安装好

执行 `emcc -v` ，如果安装好会出现如下信息：

```
emcc (Emscripten gcc/clang-like replacement + linker emulating
GNU ld) 1.37.21
clang version 4.0.0 (https://github.com/kripken/emscripten-
fastcomp-clang.git 974b55fd84ca447c4297fc3b00cefb6394571d18)
(https://github.com/kripken/emscripten-fastcomp.git
9e4ee9a67c3b67239bd1438e31263e2e86653db5) (emscripten 1.37.21 :
1.37.21)
Target: x86_64-apple-darwin15.5.0
Thread model: posix
InstalledDir: /Users/magicly/emsdk-
portable/clang/fastcomp/build_incoming_64/bin
INFO:root:(Emscripten: Running sanity checks)
```

Hello, WebAssembly!

创建一个文件 `hello.c` ：

```
#include <stdio.h>
int main() {
  printf("Hello, WebAssembly!\n");
  return 0;
}
```

编译 C/C++ 代码：

为了让代码运行在网页里面，执行下面命令会生成 hello.html 和 hello.js 两个文件，其中 hello.js 和 a.out.js 内容是完全一样的。

```
emcc hello.c -o hello.html
```

```
➜  webasm-study md5 a.out.js
MD5 (a.out.js) = d7397f44f817526a4d0f94bc85e46429
➜  webasm-study md5 hello.js
MD5 (hello.js) = d7397f44f817526a4d0f94bc85e46429
```

然后在浏览器打开hello.html，可以看到页面：；；

前面生成的代码都是 asm.js ，毕竟Emscripten是人家作者Alon Zakai最早用来生成 asm.js 的，默认输出 asm.js 也就不足为奇了。当然，可以通过option生成 wasm ，会生成三个文件： hello-wasm.html , hello-wasm.js , hello-wasm.wasm 。

```
emcc hello.c -s WASM=1 -o hello-wasm.html
```

然后浏览器打开 hello-wasm.html ，发现报错 TypeError: Failed to fetch 。原因是 wasm 文件是通过 XHR 异步加载的，用 file:/// 访问会报错，所以我们需要启一个服务器

在文件 add.c 中写如下代码：

```
#include <stdio.h>
int add(int a, int b) {
  return a + b;
}

int main() {
  printf("a + b: %d", add(1, 2));
  return 0;
}
```

有两种方法可以把 add 方法暴露出来给js调用。

通过命令行参数暴露API

```
emcc -s EXPORTED_FUNCTIONS="['_add']" add.c -o add.js
```

注意方法名 add 前必须加 _ 。
然后我们可以在 Node.js 里面这样使用：


```
// file node-add.js
const add_module = require('./add.js');
console.log(add_module.ccall('add', 'number', ['number',
'number'], [2, 3]));
```

执行 node node-add.js 会输出 5 。如果需要在web页面使用的话，执行：

```
emcc -s EXPORTED_FUNCTIONS="['_add']" add.c -o add.html
```

Module.ccall 会直接调用 C/C++ 代码的方法，更通用的场景是我们获取到一个包装过的函数，可以在js里面反复调用，这需要用 Module.cwrap ，具体细节可以参看文档。

```
const cAdd = add_module.cwrap('add', 'number', ['number',
'number']);
console.log(cAdd(2, 3));
console.log(cAdd(2, 4));
```

定义函数的时候添加 EMSCRIPTEN_KEEPALIVE

添加文件 add2.c 。

```
#include <stdio.h>
#include <emscripten.h>

int EMSCRIPTEN_KEEPALIVE add(int a, int b) {
  return a + b;
}

int main() {
  printf("a + b: %d", add(1, 2));
  return 0;
}
```


执行命令：

```
emcc add2.c -o add2.html
```

同样在 add2.html 中添加代码：

```
<button onclick="nativeAdd()">click</button>
```

可以通过在 main() 中添加 emscripten_exit_with_live_runtime() 解决：

```
#include <stdio.h>
#include <emscripten.h>

int EMSCRIPTEN_KEEPALIVE add(int a, int b) {
  return a + b;
}

int main() {
  printf("a + b: %d", add(1, 2));
  emscripten_exit_with_live_runtime();
  return 0;
}
```

或者也可以直接在命令行中添加 -s NO_EXIT_RUNTIME=1 来解决，

```
emcc add2.c -o add2.js -s NO_EXIT_RUNTIME=1
```

不过会报一个警告：


```
exit(0) implicitly called by end of main(), but noExitRuntime, so
not exiting the runtime (you can use emscripten_force_exit, if
you want to force a true shutdown)
```

所以建议采用第一种方法。

上述生成的代码都是 asm.js ，只需要在编译参数中添加 -s WASM=1 中就可以生成 wasm ，然后使用方法都一样。

用asm.js和WebAssembly执行耗时计算

```
  for (long i = start; i <= end; i += 3) {
    total -= i;
  }
  return total;
}

int main() {
  printf("sum(0, 1000000000): %ld", sum(0, 1000000000));
  // emscripten_exit_with_live_runtime();
  return 0;
}
```

注意用 gcc 编译的时候需要把跟 emscriten 相关的两行代码注释掉，否则编译不过。我们先直接用 gcc 编译成 native code 看看代码运行多块呢？

```
➜  webasm-study gcc sum.c
➜  webasm-study time ./a.out
sum(0, 1000000000): 0./a.out  5.70s user 0.02s system 99% cpu
5.746 total
➜  webasm-study gcc -O1 sum.c
➜  webasm-study time ./a.out
sum(0, 1000000000): 0./a.out  0.00s user 0.00s system 64% cpu
0.003 total
➜  webasm-study gcc -O2 sum.c
➜  webasm-study time ./a.out
sum(0, 1000000000): 0./a.out  0.00s user 0.00s system 64% cpu
0.003 total
```


可以看到有没有优化差别还是很大的，优化过的代码执行时间是**3ms!**。really？仔细想想，我for循环了10亿次啊，每次for执行大概是两次加法，两次赋值，一次比较，而我总共做了两次for循环，也就是说至少是100亿次操作，而我的mac pro是 2.5 GHz Intel Core i7 ，所以1s应该也就执行25亿次CPU指令操作吧，怎么可能逆天到这种程度，肯定是哪里错了。想起之前看到的一篇rust测试性能的文章，说rust直接在编译的时候算出了答案， 然后把结果直接写到了编译出来的代码里， 不知道gcc是不是也做了类似的事

```
    } else if (i % 5 == 0 || i % 7 == 1) {
      total += i / 2;
    }
  }
  for (long i = start; i <= end; i += 1) {
    if (i % 2 == 0 || i % 3 == 1) {
      total -= i;
    } else if (i % 5 == 0 || i % 7 == 1) {
      total -= i / 2;
    }
  }
  return total;
}

int main() {
  printf("sum(0, 1000000000): %ld", sum(0, 100000000));
  // emscripten_exit_with_live_runtime();
  return 0;
}
```

执行结果大概要正常一些了。

```
➜  webasm-study gcc -O2 sum.c
➜  webasm-study time ./a.out
sum(0, 1000000000): 0./a.out  0.32s user 0.00s system 99% cpu
0.324 total
```


ok，我们来编译成 asm.js 了。

```
#include <stdio.h>
#include <emscripten.h>

long EMSCRIPTEN_KEEPALIVE sum(long start, long end) {
// long sum(long start, long end) {
```

```
  return total;
}

int main() {
  printf("sum(0, 1000000000): %ld", sum(0, 100000000));
  emscripten_exit_with_live_runtime();
  return 0;
}
```

执行：

```
emcc sum.c -o sum.html
```

然后在 sum.html 中添加代码

```
  <button onclick="nativeSum()">NativeSum</button>
  <button onclick="jsSumCalc()">JSSum</button>
  <script type='text/javascript'>
    function nativeSum() {
      t1 = Date.now();
      const result = Module.ccall('sum', 'number', ['number',
'number'], [0, 100000000]);
      t2 = Date.now();
      console.log(`result: ${result}, cost time: ${t2 - t1}`);
    }
  </script>
  <script type='text/javascript'>
    function jsSum(start, end) {
      let total = 0;
      for (let i = start; i <= end; i += 1) {
        if (i % 2 == 0 || i % 3 == 1) {
          total += i;
        } else if (i % 5 == 0 || i % 7 == 1) {
          total += i / 2;
```

```
        result = jsSum(0, N);
        t2 = Date.now();
        console.log(`result: ${result}, cost time: ${t2 - t1}`);
      }
    </script>
```

另外，我们修改成编译成WebAssembly看看效果呢？

```
emcc sum.c -o sum.js -s WASM=1
```

Browser	webassembly	asm.js	js
Chrome61	1300ms	600ms	3300ms
Firefox55	600ms	800ms	700ms
Safari9.1	不支持	2800ms	因不支持ES6我懒得改写没测试

感觉Firefox有点不合理啊， 默认的JS太强了吧。然后觉得webassembly也没有特别强啊，突然发现 `emcc` 编译的时候没有指定优化选项 `-O2` 。再来一次：

```
emcc -O2 sum.c -o sum.js # for asm.js
emcc -O2 sum.c -o sum.js -s WASM=1 # for webassembly
```


Browser	webassembly -O2	asm.js -O2	js
Chrome61	1300ms	600ms	3300ms
Firefox55	650ms	630ms	700ms

居然没什么变化， 大失所望。号称 `asm.js` 可以达到native的50%速度么，这个倒是好像达到了。但是今年 Compiling for the Web with WebAssembly (Google I/O '17) 里说

- http://www.ruanyifeng.com/blog/2017/09/asmjs_emscripten.html
- https://zhuanlan.zhihu.com/p/25865972

前端高性能计算之三：GPU加速计算

人工智能是最近两年绝对的热点，而这次人工智能的复兴，有一个很重要的原因就是计算能力的提升，主要依赖于GPU。去年Nvidia的股价飙升了几倍，市面上好点的GPU一般都买不到，因为全被做深度学习以及挖比特币的人买光了��。

GPU，全称Graphics Processing Unit，即图像处理器，早期主要用于显示图像使用。因为图像处理主要偏简单的矩阵运算，逻辑判断等很少，因此GPU的设计跟CPU架构不一样，也因此做到一个GPU上可以有很多计算单元，可以进行大量并行计算。网上找到一个视频，应该是Nvidia某年的产品发布会，形象地演示了CPU跟GPU的区别。http://v.youku.com/v_show/id_XNDcyNTc1MjQ4==.html 。知乎上也有对CPU和GPU的对比 https://www.zhihu.com/question/19903344。

后来人们逐渐发现，GPU的这种特性还可以用于神经网络的训练，因为神经网络训练中也是大量的矩阵运算，然后原来的训练速度提高了几十倍，原来需要一周训练的模型，现在几个小时就可以出结果，于是神经网络飞速发展。。。

GPU虽快，但是写起来很难写，要用自己特殊的语言GLSL - OpenGL Shading Language编写，一般都是是将其它语言编译过来或者有很多库封装好了直接使用。经过搜索发现了gpu.js这个库。

> gpu.js is a JavaScript library for GPGPU (General purpose computing on GPUs) in the browser. gpu.js will automatically compile specially written JavaScript functions into shader language and run them on the GPU using the WebGL API. In case WebGL is not available, the functions will still run in regular JavaScript.

BENCHMARK IT!

TEXTURE MODE (GPU ONLY)

- (•) Disabled (default)
- ( ) Enabled (This is where the REAL POWER IS!)

CPU: 0.465s ±2.8%

GPU: 0.103s ±3.3% *(4.50 times faster!)*

Benchmarks provided by benchmark.js

▶ Run Benchmark

BENCHMARK IT!

TEXTURE MODE (GPU ONLY)

- ( ) Disabled (default)
- (•) Enabled (This is where the REAL POWER IS!)

CPU: 0.401s ±4.4%

GPU: 0.036s ±155.1% *(11.21 times faster!)*

Benchmarks provided by benchmark.js

- Javascript Math functions (Math.floor() and etc.)
- Loops
- if and else statements
- const and let
- No variables captured by a closure

Show Me Code

https://github.com/abhisheksoni27/gpu.js-demo

```
const c = document.getElementById('c');

const gpu = new GPU({
    mode: 'gpu'
});

// Generate Matrices
const matrices = generateMatrices();
const A = matrices.A;
const B = matrices.B;

const gpuMatMult = gpu.createKernel(function (A, B) {
    var sum = 0;
    for (var i = 0; i < 512; i++) {
        sum += A[this.thread.y][i] * B[i][this.thread.x];
    }
    return sum;
})
    .setDimensions([A.length, B.length])
// .setOutputToTexture(true);

function cpuMatMult(m, n) {
    var result = [];
    for (var i = 0; i < m.length; i++) {
        result[i] = [];
```

```
        for (let j = 0; j < m[i].length; j += 1) {
            total += m[i][j];
        }
        return total;
    }
}

//CPU
const startCPU = window.performance.now();
const cpuResult = cpuMatMult(A, B);
const endCPU = window.performance.now();
const cpuTime = endCPU - startCPU;
console.log(`CPU: ${cpuTime}ms, total: ${sumMatrix(cpuResult)}`);

// //GPU
const startGPU = window.performance.now();
const result = gpuMatMult(A, B);
console.log('gpuResult: ', result);
const endGPU = window.performance.now();
const gpuTime = endGPU - startGPU;
console.log(`GPU: ${gpuTime}ms, total: ${sumMatrix(result)}`);

//Diff
const diff = (cpuTime - gpuTime) / (gpuTime);
console.log(`%c ${diff}`, 'color: red;', `times faster!`)


function generateMatrices() {
    const matSize = 512;
    let A = [];
    let B = [];
    for (let i = 0; i < matSize; i += 1) {
        A[i] = [];
        B[i] = [];
        for (let j = 0; j < matSize; j += 1) {
            A[i][j] = i * matSize + (j + 1);
            B[i][j] = i * matSize + (j + 1);
        }
```

总结

由于架构设计不一样，GPU很适合做简单的并发计算，应用于图像处理、深度学习等领域能大大加快速度，也直接引爆了这一次人工智能的发展。当然直接用gpu去开发程序很难编写，一般都是由特殊编译器将代码编译成可以在gpu上执行的代码。本文提高的[gpu.js][gpu]就是在前端将js的一个子集编译成能在webgl上执行的一个编译器。

我们的业务逻辑比较复杂，发现很难把代码改写成能在GPU上加速执行的，最后我们采用的是之前讲过的WebWorkers+WebAssembly的方式，提速也能达到数十倍，代码还简单很多，易于维护。当然不是说复杂的问题不能转化到GPU上执行，这篇文章 https://amoffat.github.io/held-karp-gpu-demo/ 就讲怎么用GPU加速去解决TSP问题，方法很巧妙，有兴趣的可以看看。

Refers

- https://github.com/gpujs
- http://gpujs.github.io/usr-docs/files/gpu-js.html
- https://hackernoon.com/introducing-gpu-js-gpu-accelerated-javascript-ba11a6069327
- http://gpu.rocks/playground/
- https://github.com/PAIR-code/deeplearnjs
- https://deeplearnjs.org/
- https://amoffat.github.io/held-karp-gpu-demo/
- https://github.com/turbo/js
- https://github.com/stormcolor/webclgl

最后

一般而言，前端都是偏展示类的应用，重在交互和UI，很少有在前端做高计算密集型的