Furious.js: a Model for Offloading Compute-Intensive JavaScript Applications

Marat Dukhan Robert Guthrie
Robertson Taylor Richard Vuduc

Georgia Institute of Technology,
College of Computing

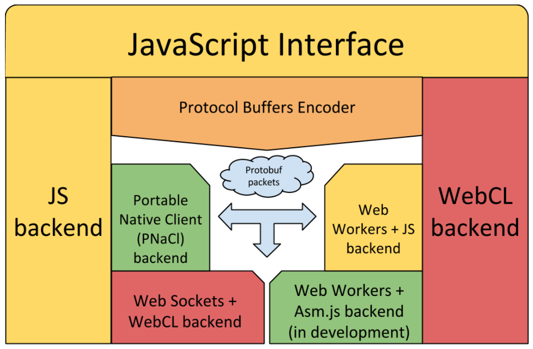
The goal of this research is to design portable hardware-accelerated interfaces for the Web. Our motivation is that Web browsers have emerged as fully-featured computing platforms that can rival traditional desktop software in functionality, yet Web applications can significantly lag desktop software in achievable performance. The as-yet unsolved problem in Web application development is how to provide near-native hardware performance portably across Web browser platforms.

In this talk, we review the existing technologies that aim to deliver near-native performance on the Web. These technologies all can interface with JavaScript code, because it is the de facto programming language of the Web, being the only language with universal support across Web browsers. However, none of these technologies has universal support across Web browsers. Therefore, we are developing a software framework, Furious.js, to overcome this problem. Furious.js provides a portable interface for high-performance multidimensional array processing. It can transparently offload computations from JavaScript to an accelerated backend, by loose analogy to NumPy for Python programs.

Furious.js is designed to overcome several of the unique limitations of JavaScript-powered Web development relative to desktop development. For example, normally a JavaScript program executes in the main UI thread. All Web app events are processed on this thread; thus, if some script runs for long time, it stalls UI processing, which distracts the user. To let programmers do intensive computations in the background, most browsers support Web Workers, a JavaScript analog of threads or processes. However, unlike threads in desktop environments, Web Workers cannot share any data: they can communicate with each other and the main UI thread only by sending and receiving asynchronous messages. Another challenge is emerging as scientific computing applications migrate to the Web. These applications often want more performance than can be delivered by JavaScript interpreters and JIT compilers. Browser developers are aware of this performance need and have suggested several technologies that can improve performance. All of these technologies work essentially by exposing low-level hardware features. Unfortunately, these extensions are mostly non-portable, with each being supported on some Web browsers but not others. These are among the challenges Furious.js aims to overcome.

To elaborate on how, consider several of the existing technologies that aim to improve the performance possible from JavaScript. First, there is Mozilla's Asm.js. It is a subset of JavaScript with special syntax that makes it, in fact, statically typed, while remaining conformant with the language specification. Asm.js code is commonly generated by compiling C or C++ code with the Emscripten cross-compiler. Mozilla Firefox and other Gecko-based browsers recognize the special structure of Asm.js code and compile it ahead-of-time as statically typed language. Another technology is Portable Native Client (PNaCl). PNaCL supported by Google Chrome and other Chromium-based Web Browsers. In PNaCl a C or C++ program is compiled into portable bitcode and in this form deployed on a Web server. When a Chromium-based browser loads PNaCl bitcode, it translates the bitcode into architecture-specific machine code, that safely runs inside a sandbox. Like Asm.js, Portable Native Client is statically typed, but it has additional benefits: use of shared-memory multithreading, efficient 64-bit operations and low-level instructions, and limited SIMD capabilities. The most powerful technology, WebCL, maps OpenCL API onto JavaScript, enabling Web pages and applications do general-purpose computations on GPUs, in addition to exposing multi-core and SIMD capabilities of CPUs. However, the WebCL standard is very new and, at the time of the writing, supported only by experimental versions of Web browsers.

Furious.js is the first JavaScript library that can use JavaScript Web Worker, Portable Native Client, WebCL, or cloud computing backend to accelerate computations. Furious.js works on both desktop and mobile browsers, and can even be used to accelerate server-side JavaScript. The library provides n-dimensional array class and a set of operations on it, such as addition, arithmetic sequence, generalized dot product, or Cholesky decomposition. Furious.js schedules computations asynchronously, but maintains an abstraction of immediate execution: computational methods immediately return a new array object that can be used for further operations. Only the final data retrieve involves an asynchronous call. Thus, the library avoid blocking the UI thread, but provides the users with conventional mostly synchronous API. To simplify code porting and library adaptation, the function names and arguments in Furious.js are kept similar to popular numpy library for Python.


JavaScript Interface
Protocol Buffers Encoder
JS backend
Protobuf packets
Portable Native Client (PNaCl) backend
Web Workers + JS backend
WebCL backend
Web Sockets + WebCL backend
Web Workers + Asm.js backend (in development)

Internally, Furious.js consists of front-end classes, which are visible to the user, and several interchangeable backends. When the library is initialized, it automatically detects the environment features and chooses the optimal backend, albeit it is also possible to specify the backend manually. A simple JavaScript backend and the WebCL backend are directly connected to the front-end interfaces: when user calls the front-end methods they are simply redirected to corresponding methods of these backends. However, this approach can not work for Web Worker or PNaCl because in these cases the computational thread lives in its own address space and can communicate with the Furious.js front-end only through asynchronous messages. Similarly, when computations are offloaded to a cloud, communication is possible only through asynchronous network connections. Thus, for these backends, the front-end serializes computational commands and input data into binary messages according to Protocol Buffers standard. Then these messages are transmitted to backend, de-serialized, and executed in the target process.

A tricky issue in Furious.js is memory management. The user-visible NDArray objects live in managed, garbage-collected JavaScript heap, but they can logically own objects in Asm.js, PNaCl, or WebCL memory that need to be manually deallocated. Unfortunately, JavaScript provides no way to learn when an object is being destroyed to deallocate its associated resources. To resolve this issue, Furious.js adopted a convention, that all NDArrays are reference-counted and every use of an array decrements its reference count. When reference count reach zero, the object is invalidated and associated memory is deallocated. To use an NDArray multiple times, a user must either increment its reference counter, or lock an object, which protects its reference counter until the object is manually unlocked or invalidated.

Acknowledgements

This material is based upon work supported by Advanced Micro Devices (AMD) and by the U.S. National Science Foundation (NSF) under Award Number 1422935. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of AMD and NSF.