High Performance Client-Side Web Programming with SPOC and Js_of_ocaml

Mathias Bourgoin and Emmanuel Chailloux

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France `firstname.lastname@lip6.fr`

Abstract. We present WebSpoc, an OCaml GPGPU library targeting web applications that is built upon SPOC and `js_of_ocaml`. SPOC is an OCaml GPGPU library focusing on abstracting memory transfers, handling GPGPU computations and offering easy portability. `Js_of_ocaml` is the OCaml bytecode to JavaScript compiler. Thus, WebSpoc provides high performance computations from the web browser while benefitting from OCaml, and SPOC high level of abstraction to increase expressivity as well as productivity.

Keywords: Web, GPGPU, OCaml, WebCL, JavaScript, High Performance

Client-side web programming currently means using technologies embedded in web browsers to run computations on the client computer. Most solutions imply using JavaScript interpreters. Using JavaScript, it is possible to describe computations, and modifications of the HTML document displayed by the browser. However, JavaScript limits static checking as everything (types, names, etc.) is checked at runtime. This leads to complex applications development with difficult debugging. `Js_of_ocaml` [1] is an OCaml bytecode to JavaScript compiler. Thus, it benefits from OCaml compilation to bytecode to provide static type checking, detecting many JavaScript runtime errors at compile time. As OCaml is a high-level multiparadigm programming language, using `js_of_ocaml` can also help express complex applications. Besides, `js_of_ocaml` produces efficient JavaScript code and benchmarks [2] show that most of the time, OCaml programs run faster when compiled to JavaScript than with OCaml bytecode interpreter. Furthermore, JavaScript is widely portable, especially to mobile devices, and `js_of_ocaml` helps targeting these architectures.

1 A need for intensive computations

With JavaScript becoming an essential part of web browsers and mobile systems, JavaScript interpreters have become highly efficient. This has lead to more demanding web applications. Besides, full Operating Systems such as ChromeOS or FireFox OS are now based on JavaScript. It, thus, is now mandatory to provide solutions to express intensive computations with high performance, while keeping applications reactive, within web-oriented frameworks.

To enable the development of such applications, standards have emerged to access to GPUs from JavaScript. WebGL [3] brings 3D graphic acceleration to the browser, while WebCL [4] targets intensive general purpose computations, as does OpenCL. OpenCL and WebCL both offer to use GPUs (as well as multi-core CPUs) as highly efficient parallel co-processors.

In order to provide a coherent solution, making GPGPU web programming easier and safer, we propose WebSpoc, an OCaml library, dedicated to `js_of_ocaml`, based on the high-level GPGPU OCaml library, SPOC[5].

2 High Performance OCaml with SPOC

GPGPU programming is a complex field, bringing it into the browser makes it even more difficult. GPGPU programming demands to write two kinds of programs, a host program that runs on the CPU host and GPGPU kernels that run on accelerators.

SPOC provides GPGPU programming with OCaml. It is based on the Cuda and OpenCL frameworks, abstracting some of the low-level, verbose and error-prone boilerplate.

SPOC is compatible with Cuda and OpenCL. Through a unified API it offers to handle any kind of GPU indifferently and conjointly in heterogeneous multi-GPGPU systems.

The memory bandwidth of the bus linking GPGPU devices and the CPU host is relatively small (5-10% of that of GPGPUs). Transfers are thus an important source of bugs and performance loss in GPGPU software. To ease GPGPU programming, SPOC manages transfers automatically. It introduces a vector data-set that is very similar to OCaml bigarrays that are monomorphic and can contain different kinds of integers, floats or booleans. Besides, SPOC always keeps the information on vectors location (on CPU or GPGPU memory) and is able to transfer them when needed. In particular, SPOC triggers transfers when vectors needed by a kernel are still in CPU memory. Similarly, when the CPU reads or writes in a vector, SPOC checks its location and transfers it if needed. Besides, SPOC deeply relies on the OCaml garbage collector to free vectors on GPU memory when they become useless to the program.

To express kernels SPOC offers a domain specific language, embedded into OCaml: Sarek. It has been built as a CamlP4 OCaml syntax extension. It features an OCaml-like syntax with type inference and static type checking. Current solutions are very difficult to debug, especially with OpenCL (and thus WebCL) which compiles GPGPU kernels at runtime. Through static type-checking Sarek, already improves GPGPU programming by enabling error detection at compile-time. Otherwise, Sarek is very similar to the C subsets that are offered in Cuda or OpenCL. It is an imperative language used to express monomorphic elementary operations that will automatically be computed in parallel by the multiple computation units of GPGPU devices.

3 Gluing the pieces : a supercomputer in your browser

The SPOC library consists of two separated code, C code, using Cuda and OpenCL APIs to handle GPGPU programming and OCaml code, managing the C functions to provide higher level features. Using `js_of_ocaml`, it is possible to keep most of the OCaml code untouched, directly benefitting from SPOC abstractions in the browser. However, it is still mandatory to provide JavaScript functions, corresponding to the C functions of SPOC. While beeing straightforward, this demanded to take care of the internal representation of ocaml values in JavaScript as well as adapt the low-level code to WebCL that is object-oriented while OpenCL is sequential. Besides, several OpenCL features have not been ported to WebCL forcing us to adapt WebSpoc in consequence. In particular, in most implementations, WebCL events are not implemented, making difficult the use of efficient asynchronous routines as in SPOC. Furthermore, multiple WebCL implementations exist, they are young, unstable and incompatible as the standard has been completed only recently. This makes managing WebCL applications complex, especially when focusing on portability.

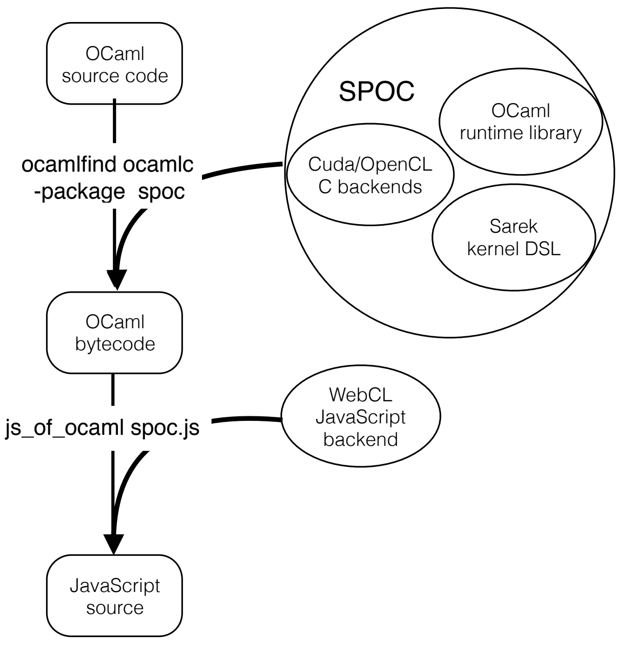

Fig. 1: WebSpoc compilation scheme

Figure 1 presents the compilation scheme of a program using WebSpoc. Using SPOC, it is compiled from OCaml to Ocaml bytecode. This bytecode can be used directly *via* the OCaml bytecode interpreter, as SPOC is dynamically linked to Cuda and OpenCL. However, using `js_of_ocaml`, it is then transformed to JavaScript code, that cannot access SPOC low-level backends. On the web page, these functions are provided via WebSpoc JavaScript low-level library. The generated JavaScript application is then able to use GPUS and multi-core CPUs through WebCL.

WebSpoc offers most SPOC features : automatic device detection, automatic memory transfers, kernel expression *via* Sarek, as well as static type checking of the whole program with dynamic compilation of GPGPU kernels to OpenCL. However, as SPOC is relying on the OCaml garbage collector to manage vectors, it is necessary to implement such a mechanism with WebSpoc. Currently, most browsers provide their own memory manager, which most of the time is inaccessible from JavaScript, and `js_of_ocaml` generates JavaScript code relying on the browser memory manager. The current WebSpoc implementation does not provide its own garbage collector, thus vector are automatically managed by the JavaScript interpreter, which means that they are automatically allocated on both memory (CPU or GPU) but the lack of primitive to access to the memory manager, prevents us (in some occasions) from providing automatic de-allocation on the GPU side.

Besides, using SPOC with web browsers makes difficult the use of external OpenCL kernels that have to be provided as external source files and compiled at runtime by SPOC. Currently, WebSpoc only supports kernels written with Sarek, that are embedded into the OCaml code and thus into the generated JavaScript. They can be compile dynamically without looking to external resources from the web page.

```
let gpu_gray = kern v ->
  let open Std in
  let tid = thread_idx_x + block_dim_x *
                           block_idx_x in
  if tid <= (512*512) then (
    let i = (tid*4) in
    let res = int_of_float
        ((0.21 *. (float (v.[<i>]))) +.
         (0.71 *. (float (v.[<i+1>]))) +.
         (0.07 *. (float (v.[<i+2>]))) ) in
    v.[<i>] <- res;
    v.[<i+1>] <- res;
    v.[<i+2>] <- res)
```

Sarek kernel

```
let vect = Vector.create Vector.int32
     (512*512*4) in
let c = canvas##getContext(Dom_html._2d_) in
let imageData = c##getImageData
     (0., 0., 512., 512.) in
for i = 0 to Vector.length gpu_vect - 1 do
  vect.[<i>] <- Int32.of_int
       (pixel_get data i);
done;
Kirc.gen ~only:Devices.OpenCL gpu_gray;
Kirc.run gpu_gray vect 0 dev);
let data = imageData##data in
for i = 0 to Vector.length vect - 1 do
  let t = Int32.to_int vect.[<i>] in
  pixel_set data i t
done
```

OCaml host code

Fig. 2: Simple example

Figure 2 presents a simple Sarek kernel transforming a color picture rendered within an HTML5 Canvas into black and white, with some of the OCaml code needed to launch it. The kernel only describes elementary computations (over one pixel of the picture) and SPOC, associated with GPGPU frameworks tools and devices drivers will automatically map it to the many computation units of the GPU in order to provide the overall parallel computation.

As we can see in this simple example, a lot of computation time is used to translate the canvas content into a vector that SPOC and Sarek can handle, and *vice versa.* This is mainly due to current WebCL implementations beeing very young and not being able to manage Canvas or WebGL data directly. Hopefully, in the near future, WebCL will provide it and SPOC will then be extended to access those contents.

Using WebSpoc, we have been able to port several SPOC applications and compare their performance with sequential OCaml implementation, using the OCaml bytecode interpreter, `js_of_ocaml`, or native binaries.

Program	jsoo	bytecode	native	WebSpoc	SPOC
MatMul	1	0.7	6.5	129.5	189.6
Sort	1	1.5	8.3	24.3	34.0
MandelBrot	1	1	1.1	99.6	118.6

Table 1: Benchmarks (speedups) using an Nvidia GTX 680

Table 1 presents these benchmarks. We used three examples, `MatMul` that naively multiplies 4000x4000 matrices, `Sorts` that sorts integer vectors using a naive implementation of the bitonic sorting algorithm and `Mandelbrot` that computes a mandelbrot set with a large maximum number of iterations (10000). Here we used the Firefox 28 web browser with the Nokia implementation[1] of WebCL, on a Ubuntu 14.10 machine with the Nvidia OpenCL implementation coming with the Cuda 5.5 SDK. The table presents the speedups achieved with different solutions compared to an implementation written in OCaml and compiled with `js_of_ocaml`. It shows that using SPOC offloads most of the computations to the GPU and thus provides almost the performance of the native SPOC implementation. Besides, both SPOC and WebSpoc largely improve performance over JavaScript code generated with `js_of_ocaml`, bytecode interpretation and native sequential computations. The main difference between native SPOC and WebSpoc comes from the use of synchronous transfers within WebSpoc and from the loss in performance in the translation of OCaml part of the runtime generated with `js_of_ocaml`.

4 Perspectives

Using `js_of_ocaml`, it is possible to benefit from high-level features from OCaml within web frameworks. Using SPOC, OCaml programs can benefit from high performance GPUs to increase their efficiency. Mixing both naturally provides high performance with high-level abstractions in a typed environment to web frameworks. Using WebSpoc, it thus becomes possible to develop intensive applications such as rich video games or photo/video editors that runs in web browsers.

However, to fully benefit from SPOC features, WebSpoc must integrate a memory manager with a garbage collector in order to automatically free vectors on GPU when needed, without explicitly transferring them to the CPU. This memory manager could be implemented using Asm.js [6], a low-level library for JavaScript that targets high performance application and allow fine memory management. As we showed previously, WebSpoc currently lacks interoperabilty with Canvas contents and WebGL. To target most visually intensive applications, this is where our future efforts will be oriented.

Furthermore, JavaScript can be used with multiple platforms. In particular, Node.js[7], that can run JavaScript code as system applications. It comes with the necessary tools to build distributed software and servers. It could be interesting to compare GPGPU High Performance Computing software's performance running on GPGPU clusters using SPOC with classic ocaml-mpi, Async Parallel[8] and JavaScript code generated through `js_of_ocaml` running within Node.js.

References

1. J. Vouillon and V. Balat. From Bytecode to JavaScript: The js_of_ocaml Compiler. *Software: Practice and Experience*, 2013.
2. Ocsigen team. Performances of Js_of_ocaml compiled programs. `ocsigen.org/js_of_ocaml/manual/performances`.
3. D. Jackson. WebGL 2 Specification. *Khronos WebGL Working Group*, 2014.
4. T. Aarnio and M. Bourges-Sevenier. WebCL 1.0 specification. *Khronos WebCL Working Group*, 2014.
5. M. Bourgoin, E. Chailloux, and J.-L. Lamotte. Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 2013.
6. Asm.js Working Draft. *Mozilla*, 2013.
7. S. Tilkov and S. Vinoski. Node. js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6), 2010.
8. E. Stokes. Async Parallel. `https://blogs.janestreet.com/async-parallel/`.

[1] http://webcl.nokiaresearch.com/