Data Parallel Programming for the Emerging Web

Richard L. Hudson, Tatiana Shpeisman, Adam Welc, Ali-Reza Adl-Tabatabai

Intel Labs

Abstract

JavaScript is the safe and secure language used to deliver browser based client side applications. It is typically the first language a person learns and for a large number of productivity programmers working on web based client applications, the only language they use. There is tremendous pressure on these programmers to make their web based applications more visually appealing and engaging. To do this they must leverage the available hardware resources which include multiple cores and graphics processing units.

One position for how to achieve this is to abstract the hardware using languages such OpenCL. This position requires intimate knowledge of the hardware but provides ultimate control of the hardware and optimal performance out of the hardware. The cost is that the programmer must work in two programming models, the high level JavaScript programming model suitable for the productivity programmer and the low level hardware model suitable for the performance programmer. Our position is that the productivity programmer should be given a single programming model, the JavaScript model, to work in and that it should be able to utilize the available hardware parallelism without the programmer having to drop into an unfamiliar second programming model.

1. Introduction

The next round of innovative applications will require more compute power as it always has. The hardware vendors are responding, not with faster CPUs, but with multi-cores and many-cores, often combined with GPUs and other types of accelerators.

In order to utilize capabilities of these new architectures, the programming language community has responded with new ways of expressing parallelism. All the traditional application programming languages (C, C++, Java, C#, etc.) are increasing support for parallel computations in multiple different ways – through extensions (often supporting different styles of parallelism – data parallelism, task parallelism, distributed parallelism, etc.), libraries, virtual execution environments, automated parallelization tools, parallel programming patterns, and so forth. Surprisingly, while these approaches have been at least partially successful in their own application domains, their adoption into the web-oriented programming environments and scripting programming languages, used by a larger and larger number of developers, has failed almost completely.

Acceptance of the web browser as a dominant application delivery system is growing every day. Web applications are becoming richer and more complicated with each passing month. Recent developments such as HTML5 and WebGL have added new features, such as video, audio, and 2D and 3D graphics. These features provide ample opportunities for parallel client-side applications. Image and video processing, physics, financial applications could utilize client hardware resources to enable rich immersive visual experience for the user. Support for parallelism in the most popular client-side web development language – JavaScript – remains very limited. The need for task parallelism has been, at least partially, addressed by web workers – effectively, coarse grain threads that communicate via asynchronous message passing. Yet, support for data parallel programming in JavaScript is still non-existent. Data-parallel algorithms are easy to scale and are a good match to graphics and vector hardware. It is quite astonishing to us that one of the most popular programming languages has been neglected with respect to data parallelism. We hope to change this situation by providing support for data parallelism directly in JavaScript. The goal is not to achieve parity by duplicating the solutions available for traditional programming languages. Instead it is to embrace and extend JavaScript's programming model so that existing JavaScript programmers can leverage the power of modern hardware in new applications.

2. Data parallelism

We believe that JavaScript must provide a programming model that executes data parallel constructs on modern data parallel hardware while maintaining the productivity advantages of a dynamic scripting language.

Today, the main programming approach to data parallelism is to match GPU architecture with low-level programming models such as CUDA[3] and OpenCL[2]. With recent emergence of WebGL[7], it is logical to ask if WebCL – an adaptation of OpenCL for the browser environment – is the next step. While plausible on the surface, we believe that straightforward adaptation of OpenCL for the browser is unlikely to be accepted by the JavaScript community.

```
var b = Array(a.length);
for (i=0; i < a.length; i++)
  b[i] = a[i] + 1
```

a) Sequential version

```
b = a.map(function(x){return x+1;})
```

b) Parallel version

Figure 1. Simple JavaScript program

OpenCL provides a bifurcated programming model – a program consists of the host code executing on CPU and the device code executing on a "device" – GPU, CPU, co-processor (e.g., Cell), or some other accelerator. The host and device code are written in two distinct programming languages and communicate via the set of OpenCL APIs. While host program can be written in any programming language supported by the OpenCL API bindings, the device program (a kernel) is written in OpenCL C – a variation of C99 that exposes low-level details of the GPU architecture to the programmer. These details include, for example, a three-level memory hierarchy - private per-thread memory, local memory shared between threads organized in a thread group, and global memory shared by all threads. The programmer is also responsible for the explicit mapping and unmapping of device memory buffers to the CPU memory and synchronization via barriers. To write OpenCL code, a programmer should be aware of all these details; moreover, to achieve optimal performance the size of the work group must correctly match both the application and the target device's architecture.

It is hard for us to imagine how this low-level programming model could be adapted to JavaScript – a safe, object-oriented, dynamically typed programming language whose primary developer base consists of application programmers and web content developers. First, the JavaScript community already rejected the relatively well-understood lock-based shared memory programming model as too complicated and error-prone. Instead it chose to support task parallelism via asynchronous message passing. OpenCL makes the problems of data races and non-deterministic execution only worse (locks might be bad, but they are better than thread group barriers). Second, OpenCL forces programmers to think in terms of two programming models, leading to classical software engineering problems – to modify an OpenCL program one needs to modify the host logic, the device code, and the glue layer between the two. Finally, the main goal of OpenCL is to achieve optimal performance on a particular device, which contradicts the nature of web applications which are designed to run on a variety of clients, ranging from the powerful desktops with top-of-the line GPUs to mobile phones. It is unrealistic to expect web developers to tune the application performance for every possible combination of CPU and GPU found on the client hardware.

The roots of the OpenCL approach to parallel programming can be found in decades of high performance computing focused on extracting performance from the hardware and not productivity from the programmer. This approach was magnified by the nature of the hardware business where purchasing decisions are made based on results from benchmarks that are highly tuned by the hardware vendors. In addition, the high cost of parallel hardware encouraged customers to go through extra hoops to achieve maximum possible utilization from their investment.

This environment has made it difficult to successfully champion programmer productivity. Fortunately, the times are changing. As parallel hardware becomes a commodity, programmer productivity emerges as the more expensive resource and, consequently, must be favored over performance by programming models. We, thus, believe that data-parallel programming model for JavaScript should draw inspiration from high-level programming models such as NESL[8] and Google's map-reduce[9], rather than OpenCL.

The common perception is that writing parallel programs is hard. Perhaps this notion comes from confusion between writing parallel programs and writing efficient parallel programs. Figure 1 shows a simple example (adding one to all elements of an array) written two ways – a traditional sequential loop-based code (Figure 1a) and a parallel style code using the JavaScript Array map method (Figure 1b) with a kernel function. The code in Figure 1b concisely captures the parallel semantics for both the programmer and the compiler while also being shorter, clearer, and in concert with the JavaScript programming model. Besides map, JavaScript also provides many high-level array operations such as reduce and filter that appear to be parallel. Unfortunately, these operations are defined to have sequential semantics. A callback to the array operation kernel function can modify external state including the array elements, the global state, and the free variables from a closure. If the callbacks were done in parallel this would be non-deterministic. To avoid this JavaScript enforces determinism by imposing sequential semantics on array methods such as map. We propose providing a new data structure - a parallel array - along with methods that achieves determinism through functional (side effect free)

semantics. Such semantics are intuitively familiar the JavaScript programmer so the acceptance bar should not be high.

Compiling arbitrary JavaScript kernels for efficient execution on vector and graphics hardware is a non-trivial problem which we do not expect to be solved overnight. We would have to develop compiler and run-time techniques to solve a large number of challenging technical problems such as support for pointer-based data structures, finding optimal data layout, minimizing transitions between CPU and GPU execution, handling code divergence, supporting dynamic types etc. The existing JavaScript story, however, teaches us that when programmability goes first, performance follows. The existing techniques for efficient JavaScript execution such as type splitting[1] and trace-based compilation[3] have improved the JavaScript performance by the order of magnitude, and there is no reason why the same will not happen for parallel execution. Meanwhile, performance-conscious programmers will learn to write in patterns that can be parallelized by today compilers. Tools will be developed to detect missed performance opportunity. These initial modest performance gains will be the improvement that will drive programmers to create application that take advantage of data parallelism. In turn the new applications will drive the development of better compilers and tools.

3. Conclusion

Our vision of the future is a web based ecosystem where the browser is the platform of choice and JavaScript continues to be the implementation language of choice. For this future to materialize we need to bring data parallelism to JavaScript and in turn enable future innovations. Our position is that data parallelism can be achieved by starting with a parallel array type and a few simple constructs that can leverage GPUs and other accelerators. If the programmer writes code that is correct but uses constructs that cannot be parallelized it is the software development environment's job to assist the programmer but the code running in the browser is required to still run correctly and deterministically.

4. References

[1] Chambers, C. and Ungar, D. *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs.* Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, pp. 150-164, White Plains, NY, June, 1990.

[2] Khronos OpenCL Working Group, *The OpenCL Specification*. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[3] Gal A.; Eich, B.; Shaver, M.; Anderson, D; Mandelin, D.; Haghighat, M. R.; Kaplan, B.; Hoare, G.; Zbarsky, B.; Orendorff, J.; Ruderman, J.; Smith, E. W.; Reitmaier, R.; Bebenita, M.; Chang, M. and Franz, M. *Trace-based just-in-time type specialization for dynamic languages*. In Proceedings of the Conference on Programming Language Design and Implementation, pages 465–478, 2009.

[3] NVIDA corporation, *CUDA programming guide,* June 2008.

[4] Hillis, W. D. and Steele, G. L. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (Dec. 1986), 1170-1183. DOI= http://doi.acm.org/10.1145/7902.7903

[6] Hickson, I. (editor), *Web Workers Draft Recommendation — 10 September 2010* available from http://whatwg.org/ww

[7] Marrin, C. (Editor), *WebGL Specification Working Draft 10 June 2010* available from https://khronos.org/

[8] Blelloch, G. E. 1993 *Nesl: a Nested Data-Parallel Language (Version 2.6)*. Technical Report. UMI Order Number: CS-93-129., Carnegie Mellon University.

[9] Dean, D and Ghemawat, S. *MapReduce: Simplified data processing on large clusters.* In Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), pages 137–150, San Francisco, California.