CrowdCL: Web-Based Volunteer Computing with WebCL

Tommy MacWilliam
School of Engineering and Applied Sciences
Harvard University
Cambridge, Massachusetts 02138
tmacwilliam@cs.harvard.edu

Cris Cecka
School of Engineering and Applied Sciences
Harvard University
Cambridge, Massachusetts 02138
ccecka@seas.harvard.edu

***Abstract*—We present CrowdCL, an open-source framework for the rapid development of volunteer computing and OpenCL applications on the web. Drawing inspiration from existing GPU libraries like PyCUDA, CrowdCL provides an abstraction layer for WebCL aimed at reducing boilerplate and improving code readability. CrowdCL also provides developers with a framework to easily run computations in the background of a web page, which allows developers to distribute computations across a network of clients and aggregate results on a centralized server. We compare the performance of CrowdCL against serial implementations in Javascript and Java across a variety of platforms. Our benchmark results show strong promise for the web browser as a high-performance distributed computing platform.**

I. Introduction

Through volunteer computing, users can contribute otherwise idle CPU cycles to solving computationally-intensive problems. By distributing jobs across a large network of users, volunteer computing projects can utilize more computational power than would be feasible to gather locally due to cost and space constraints. Typically, however, participation in a volunteer computing project requires users to download and install an additional package on their machine to run in the background. However, the overhead associated with the installation process may dissuade some users from participating, and users may forget to launch or inadvertently quit the background process, which also lowers participation rates. In addition, the cost of supporting a seemingly infinite set of achitectures and configurations can be significant [7].

In this paper, we introduce CrowdCL, an open-source, web-based, cross-platform, high-performance volunteer computing framework. CrowdCL allows developers to write high-performance web applications that can be executed as background processes within a web page. For example, as a user browses a website, Javascript code that finds solutions to an optimization problem can be executed by the web browser without disrupting the user's experience on the site. This avoids the need for active installation on users' computers and allows developers to create and distribute cross-platform volunteer computing applications that take advantage of more and more ubiquitous hardware accelerators.

For this purpose, CrowdCL also provides simple and adaptable utilities to take advantage of GPU computing through WebCL [3], a Javascript binding to OpenCL. To reduce the complexity and boilerplate of traditional OpenCL programs, we develop the KernelContext, a WebCL abstraction that greatly simplifies the process of communicating with the GPU, defining OpenCL programs, and launching device computations. Using both CrowdCL and KernelContext, developers can rapidly write parallel and cross-platform programs that can be distributed to and executed by a collection of volunteers.

Ultimately, the goals of CrowdCL are similar those of existing volunteer computing projects. One such effort is Folding@home [7], which runs protein folding computations as background processes on participants' computers. The project currently sustains more than 5 PetaFLOPS as a result of the contributions of an estimated 400,000 clients. Similarly, the BOINC project that powers SETI@home [5] has reached hundreds of TeraFLOPS through the participation of 500,000 users. Like BOINC, CrowdCL seeks to provide a generalized framework that can be used to solve a variety of computation problems. While Folding@home and SETI@home run as desktop applications, efforts such as Mechanical Turk [11] have used the web as a platform for crowdsourcing. While Mechanical Turk is aimed primarily at consumers, as users can create or solve micro-tasks in exchange for payment, CrowdCL instead targets developers who seek to distribute GPU computations via embedded web applications.

In this paper, we will show that the CrowdCL framework allows simplified development of distributed, OpenCL accelerated applications. These applications computationally outperform serial implementations in Javascript and Java on a single machine, and can be embedded on any web page using a single `<script>` tag in a page's HTML source.

The main contributions of this paper are:

- We develop a PyCUDA-like abstraction layer for WebCL, called KernelContext, to ease the construction and execution of OpenCL programs in an online Javascript environment.
- We compare the performance of this abstraction layer against other common cross-platform computing strategies.
- Our implementation of a CrowdCL framework for distributing the web applications allows for trivial deployment of a high-performance web application.

We use the Thomson problem as a benchmark and demonstration of our framework. The Thomson problem is a nonlinear optimization problem that is useful in many problems in biology, math, physics, and computer science. See Section IV-A.

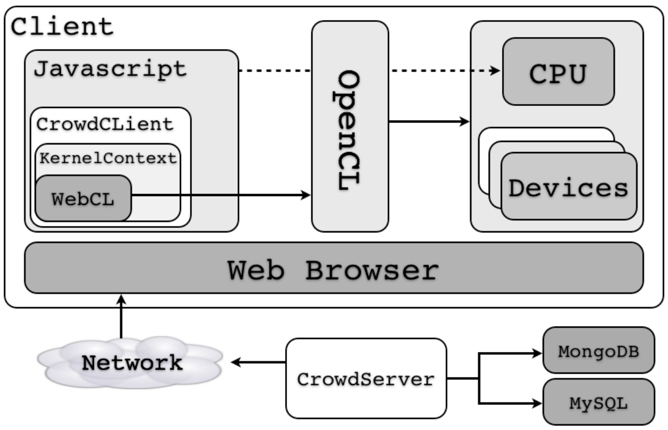

Fig. 1. With the CrowdCL framework, developers can easily integrate OpenCL code into Javascript in order to solve arbitrary computational problems via a distributed network of clients and collect results on a central server.

Let $\omega_N = x_1, \ldots, x_N$ denote a set of N distinct points in $\mathbb{S}^2 \subset \mathbb{R}^3$. For real $s > 0$, the s-energy is given by

$$E_s(\omega_N) = \sum_{\substack{x,y \in \omega_N \\ x \neq y}} \frac{1}{|x-y|^s} \tag{1}$$

For given N, the Thomson problem is then to find

$$T_N = \inf_{\omega_N} E_2(\omega_N) \tag{2}$$

Optimization procedures for the Thomson problem and related problems are computationally intense as computing E_s and $\partial_\omega E_s$ require $\mathcal{O}(N^2)$ operations. For this purpose, we wish to use hardware accelerators on the client to increase the computational intensity.

II. KernelContext

Within the CrowdCL project, the WebCL KernelContext [2] is an open-source abstraction layer for WebCL, which allows OpenCL code to be executed on the web via Javascript. The KernelContext library consists of two components: the core module and the utilities module. The KernelContext core defines methods for compiling OpenCL kernels, reading and writing GPU data, and launching kernels, while KernelContext utilities include methods for dynamically constructing and launching map/reduce kernels.

Kernels are first-class citizens in the KernelContext, and our library's API draws inspiration from existing frameworks. This API closely resembles that of PyCUDA [12], a Python framework for compiling and executing CUDA code. In particular, both PyCUDA and CrowdCL allow typed arrays to be transferred to and from the GPU using a single method. Furthermore, just as PyCUDA compiles CUDA source from strings and returns Python functions that can be called with appropriate parameters in order to launch kernels, CrowdCL uses a proxy pattern in order to launch OpenCL kernels compiled from Javascript strings via the same syntax used to call Javascript functions.

A. KernelContext Core

The KernelContext Core provides an abstraction for basic GPU operations. A connection to the GPU can be established by creating a new `KernelContext` object, which will manage all communication with the device. Data can be sent to the GPU via the `toGPU` method, which takes as input an array to send to the device and returns a handle to the data on the GPU. This handle serves as a pointer to the data on the GPU and can be used to reference GPU memory when passing arguments to a kernel or transferring data from the device to the host. To improve performance and give developers more fine-grained control over the bytes that are transferred to the device, KernelContext mandates the use of Javascript typed arrays, such as `Float32Array` and `Uint32Array`, rather than simply `Array` instances. Typed arrays can be accessed using the same syntax as `Array` instances, though typed arrays have a fixed size (and type), and not all `Array` methods are available.

The KernelContext `compile` method compiles an OpenCL program and returns a callable function. An OpenCL program can be defined as a Javascript string, placed in a separate `.js` file, or contained within HTML via the `<script type="text/x-opencl">` tag. When the function returned by the `compile` method is called, the kernel will be launched on the GPU. This function is thus a proxy object for the compiled OpenCL kernel, which allows an OpenCL kernel to be launched using the same syntax for executing Javascript function, which significantly reduces boilerplate while improving readability. When calling the kernel proxy, the first parameter must be an object that defines the global work size and local work size for the kernel via keys named `global` and `local`, respectively. Parameters to the kernel itself can also be passed to a compiled function. Pointers to arrays are represented by the handles returned by the `toGPU` method. Scalar types can also be passed to kernels via instantiations of `Uint32`, `Float32`, etc. See Code 1.

Data can be transferred from the device to the host via the `fromGPU` method, which takes as input a handle to data on the GPU obtained via a previous call to `toGPU`. In order to improve performance, a pre-allocated typed array buffer to be filled with data from the GPU may be passed as a second parameter. If a second argument is not passed, then the KernelContext will dynamically allocate a new typed array buffer to store the GPU data and return it to the caller.

```
var data = new Uint32Array(10);

var ctx = new KernelContext;
var kernel = ctx.compile(source_str, 'fn_name');

var d_data = ctx.toGPU(data);
kernel({local: 32, global: 32}, d_data);
ctx.fromGPU(d_data, data);
```

Code 1. The KernelContext can be used to send data to the GPU, compile and launch a OpenCL program, and read data from the GPU.

The KernelContext utilizes OpenCL command queues by lazily executing data transfer calls and kernel launches. For example, `toGPU` and kernel evaluations will not be evaluated until a call to `fromGPU` that depends on their execution is called. Queuing methods reduces communication with the device, resulting in higher performance for KernelContext applications. Figure 1 demonstrates KernelContext usage by sending data to the GPU, launching a kernel, and reading data from the GPU.

B. KernelContext Utilities

The KernelContext utilities class, `KernelUtils`, builds on the KernelContext core. Because many operations are common in parallel programs, `KernelUtils` provides methods to dynamically generate kernels rather than writing their source manually.

The `map` method will compile a new kernel object, transfer necessary data to the GPU, launch the kernel, transfer the result from the GPU, and return a typed array representing the result of the map. Global and local work sizes will be determined appropriately such that the mapping operation for each element will be performed in a separate thread. This method takes at least three parameters: a string containing a comma-separated list of variable names, a string containing an OpenCL snippet that represents the mapping operation, and one or more typed arrays to use in the map. Code 2 uses the `map` method to add 1 to each element of a typed array in parallel on the GPU.

```
var ctx = new KernelContext;
var util = new KernelUtils(ctx);

var a1 = new Uint32Array(10);
var result = util.map('x', 'x[i] + 1', a1);
```

Code 2. A kernel to add `1` to each element of the array `a1` is generated, compiled, and launched. Because the variable name `x` has been specified in the first argument, `x` can be used in the second argument to refer to the Javascript typed array `a1` inside the mapping kernel source.

Like `map`, the `reduce` method will compile a new kernel object, transfer necessary data to the GPU, launch the kernel, transfer the result from the GPU, and return a scalar representing the result of the reduction. The parallel reduce is performed in two phases. First, the input array is partitioned into groups based on the GPU's local work size, and the reduction is performed over each of those groups in parallel to yield one scalar per group. In the second phase, the reduction is performed over the remaining scalars to obtain a scalar result. This method has two required parameters: a string containing an OpenCL snippet that represents the reduction operation and one typed array to reduce over. Code 3 uses the `reduce` method to compute the sum and maximum value of a typed array in parallel on the GPU.

```
var a2 = new Uint32Array(10);
var sum = util.reduce('a + b', a2);
var max = util.reduce('(a > b) ? a : b', a2);
```

Code 3. Kernels to compute the sum and maximum of the array `a2` are generated, compiled, and launched. In the first argument, the OpenCL variable `a` represents the current value of the reduction accumulator, and the variable `b` represents the next value to be reduced in the array.

As the `map` and `reduce` methods return the result of the operation rather than a callable kernel, it may be desirable to instead re-use a map or reduce kernel at a later point in a program. In this case, the `mapKernel` and `reductionKernel` methods, which return a proxy object that can be called to execute the kernel rather than launching the kernel immediately, can be used. See Code 4. Whereas `map` and `reduce` will recompile and execute a new kernel each time, `mapKernel` and `reduceKernel` return a re-usable kernel proxy, similar in spirit to the KernelContext `compile` method.

```
var sum_kernel =
 util.reduceKernel(Uint32Array, 'a + b');
var max_kernel =
 util.reduceKernel(Uint32Array, '(a > b) ? a : b');
var d_a2 = ctx.toGPU(a2);
var sum2 = sum_kernel(d_a2);
var max2 = max_kernel(d_a2);
```

Code 4. Alternative use of utility kernels. Precompile the reductions kernels to be used repeatedly in the code later. Note the intuitive and easy-to-use interface that avoids complexities in writing and using a reduce kernel: launching with carefully chosen local and global thread group sizes, multiple kernel invocations for large arrays, etc.

III. CrowdCL

While the KernelContext library creates an abstraction layer for accessing the GPU via Javascript in the web browser, CrowdCL [1] provides an open-source volunteer computing framework. CrowdCL consists of a client and server component. Using the CrowdCL client framework, developers can write Javascript code to be run in the background of users' web browsers, and the results of those processes are collected using the CrowdCL server application. Per Figure 1, developers can use our framework to easily write OpenCL applications using Javascript, distribute computations to web browser clients, and aggregate results on a server.

A. CrowdCLient

CrowdCLient, the CrowdCL client library, handles the execution of arbitrary code in the background of a web page. To use the CrowdCL framework, a developer first creates a Javascript class that represents an instance of an arbitrary computational problem to be solved using volunteer computing. The constructor for the problem class can accept any number of parameters and may also perform any logic necessary to instantiate the problem instance. This class must, at a minimum, also define a `run` method, which will be executed in a loop in the background of a user's web browser by CrowdCL. The `run` method contains the logic needed to obtain a single result or solution to the problem; in the context of an optimization problem, for example, the `run` method represents a single step in the solver.

An object that has defined at least a `run` method can then be passed to the constructor of the `CrowdCLient`. The CrowdCLient instance is the driver for the developer's problem class that will handle the execution of a problem's `run` method in the background; once a CrowdCLient has been created, it will begin executing the problem's `run` method. The CrowdCLient will then send the results obtained by the problem instance to the CrowdServer via an Ajax request. If the problem instance defines a `sync` method, then this behavior can be overridden.

The CrowdCLient constructor can take a number of parameters: `id` is a string representing a unique ID for the problem, `server` is the URL of the CrowdCL server to which all computation results will be sent, `timeout` is maximum delay between consecutive calls of the `run` method, and `stage` is the number of `run` results that will be cached locally before a batch of results is sent to the CrowdCL server to be saved. A callback to be executed after every iteration of the `run` method can also be specified using the `onResult` key.

The CrowdCLient object also defines a number of events that can be used to interact with the driver. The `pause` method can be used to pause the execution of the run loop, the `resume` method can be used to resume execution, and the `sleep` method can be used to pause execution for a fixed number of milliseconds. When the execution of the `run` loop is interrupted by any of these events, the problem's `interrupt` method will be called (if it is defined), allowing the problem developer to respond to driver events (i.e., to clean up memory).

B. *CrowdServer*

The CrowdServer is a RESTful Node.js application responsible for aggregating the results of users running CrowdCL client code. By default, the CrowdServer uses MongoDB as a data store, as a schema-less database is more conducive to storing arbitrary data that may be posted by clients. However, the server also includes a MySQL adapter if MongoDB is unavailable. When using MySQL a column must exist in the MySQL table for each key that is posted by a client.

The CrowdServer has API routes for saving new results from clients as well as aggregating existing results. When a CrowdCLient has a new result (or set of results), a POST request containing a JSON object is sent to `/result/:id`, where `id` is the unique ID for the problem specified in the CrowdCLient constructor. This JSON object may contain an arbitrary set of key/value pairs, allowing developers to save as much data as desired from the problem instance. To view existing results, a GET request can be sent to `/results/:id`, where `id` is again the unique ID for the problem. Each result is also given a globally-unique identifier, and sending a GET request to the route `/result/:result_id` can be used to obtain a JSON representation of a single result.

In order to ensure fake data is not sent to and stored by the CrowdServer, the developer of a problem can specify a `verify` method. Given a single result from the data store, this method returns true if the data fits the constraints of the problem and false if the data has been fabricated by a malicious user. In the event data is found to be invalid, it is automatically removed from the data store. The `verify` method is securely executed on the CrowdServer using the Node.js `vm` sandbox in order to ensure validity.

IV. Results

We use the Thomson problem, an optimization problem, to benchmark our KernelContext and CrowdCL implementations.

A. *Benchmark – Thomson Problem*

The Thomson problem described in equation (2) remains a subject of considerable interest from its proposal as the "plum pudding" model of the atom by J.J. Thomson in the early 20th century [16]. It has applications to such diverse fields as condensed matter physics, especially crystallography, materials science [13], [6], viral morphology [14], supramolecular chemistry, global positioning, encryption problems in computer science [9], and even the design of golf balls.

The most straightforward solution to the Thomson places N charges on the sphere and uses some optimization algorithm to minimize the energy of the charges to find the solution. Of course, such approach has been tried [15], [4] but in general does not provide the actual minimum configuration because there are many configurations that are a minimum of the energy and among this many configurations we only seek the one that gives the lowest energy. In fact, Erber and Hockney [10] showed that the number of solutions to the Thomson problem grows exponentially with the number of particles, and already for number of particles $N \approx 100$ there are thousands of solutions. With so many solutions available, it becomes an enormously expensive task to find the one that actually has the lowest energy.

Because the Thomson problem is already associated with online volunteer computing efforts [8] and its computation is expensive, we use it as a proof-of-concept for the CrowdCL distributed heterogeneous framework and as a benchmark for the KernelContext library for WebCL.

The optimization algorithm used for the Thomson problem is a steepest decent with heuristic step length. The steps of the algorithm are then:

- Compute the gradient (force on each point $x \in \omega_N$),
$$G(\omega_N)[x] = \sum_{\substack{y \in \omega_N \\ y \neq x}} \frac{x - y}{|x - y|^3}. \tag{3}$$
- Compute the heuristic step-length,
$$ds := f(\omega_N, G(\omega_N)). \tag{4}$$
- Update all points in ω_N,
$$x := x + ds \cdot G(\omega_N)[x]. \tag{5}$$
- Renormalize all points to the sphere,
$$x := \frac{x}{|x|}. \tag{6}$$

Thus, this problem requires a custom N-body computation (3), mapping operations (5) (6), and a reduction operation (4). The KernelContext core and utility libraries for WebCL allow this algorithm to be written as an accelerated, cross-platform web application very straightforwardly.

B. *Performance and Comparisons*

We compare the performance of our CrowdCL implementation with a comparable implementation in raw Javascript and an optimized Java implementation on two different platforms. Keeping in mind that the application can be run passively in the background or actively in the foreground to augment user experience and computational power and does not require architecture dependent implementations or active user installation (in contrast to existing projects like Folding@home [7]), we find that this compute model for large scale distributed computation and data analysis shows strong promise.

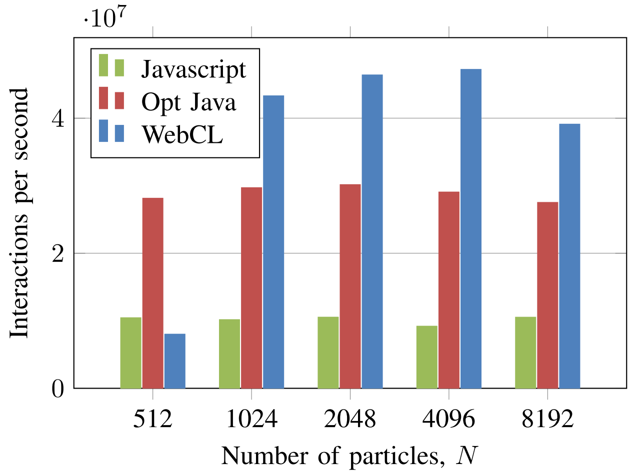

Fig. 2. On a Macbook Air with Nvidia 320M GPU, CrowdCL outperforms serial Javascript and optimized Java for $N \geq 1024$.

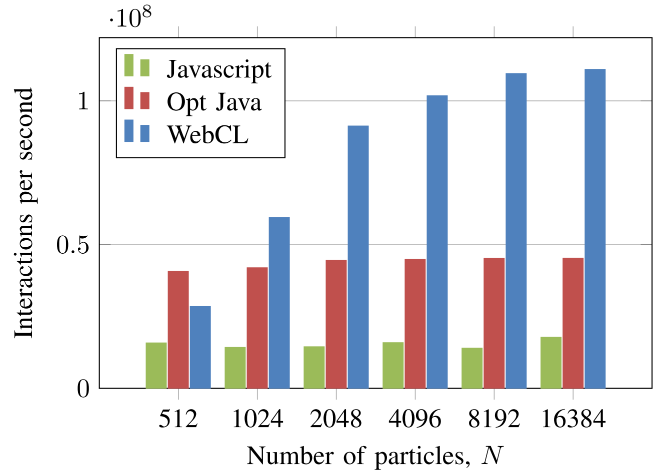

Fig. 3. On a desktop equipped with a NVIDIA Tesla K20, CrowdCL outperforms serial Javascript and optimized Java for $N \geq 512$ and the performance gap between serial and parallel implementations is larger compared to the Macbook Air.

Our CrowdCL implementation minimizes the transfer of data between the host and device by performing all computations on the GPU. The energy (1) of a given configuration of points as well as the forces (3) exerted on all points are computed entirely on the GPU. Similarly, the placement of points is updated on the device with each optimization step so that data need not be transferred on each iteration of the optimization.

We benchmark our implementation for $512 \leq N \leq 8192$ on a machine that could be owned by a typical volunteer computing user: a Macbook Air equipped with 1.86 GHz Intel Core 2 Duo CPU and an NVIDIA 320M GPU. Our results are shown in Figure 2. For $N < 1024$, serial implementations in Javascript and Java outperformed CrowdCL due to the overhead incurred by transferring data to and from the GPU and the inability to saturate the compute device with work. Because the number of operations increases as $\mathcal{O}(N^2)$ while the memory requirements increase as $\mathcal{O}(N)$, the CrowdCL implementation is more efficient for larger values of N. For all values of N Java (a compiled language) outperformed Javascript (an interpreted language).

We also benchmark our implementation for $512 \leq N \leq 16384$ using a desktop computer equipped with a Intel Xeon W3670 3.2GHz CPU and an NVIDIA Tesla K20 GPU. The results are shown in Figure 3. Every implementation is nearly an order of magnitude faster with the improved hardware. The K20 GPU significantly improved the performance of the parallel implementation, such that our CrowdCL implementation outperformed both serial implementations for $N > 512$, a lower threshold than that on the Macbook Air. Additionally, we can see that even with 16K particles, the K20 is still not saturated in computational power. Otherwise, the relative performance of all implementations remained relatively unchanged; as in the first benchmark, the optimized Java outperformed Javascript in all cases.

Finally, we benchmark the performance of the individual WebCL kernels used in our CrowdCL Thomson problem implementation. The force kernel is an N-body computation (3), the update kernel is a mapping operation (5), and the reduction kernel is a reduce operation (4). Each of these kernels is executed once for each step in the optimization over all N points on the sphere. Our results on the Macbook Air are shown in Figure 4. For small values of N, the performance of the three kernels is comparable. However, as N increases, the N-body computation is the most efficient kernel, while the reduction kernel achieves the lowest performance.

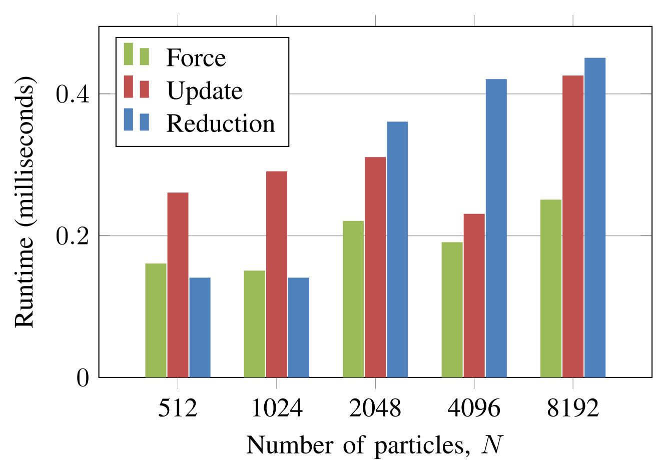

Fig. 4. On the Macbook Air, the performance of the three main kernels—computing forces on the sphere, updating the positions of points on the sphere, and performing a reduction across all points—in our Thomson problem solver were comparable for small values of N. For larger values, the reduction kernel had the lowest performance, while the N-body kernel had the highest performance.

V. Conclusion

The WebCL KernelContext and CrowdCL volunteer computing framework simplify the development of web-based GPU applications as well as the distribution of computational tasks across a network of users. The WebCL KernelContext allows developers to rapidly create OpenCL applications using Javascript by providing a PyCUDA-like API for launching GPU computations. Building on the KernelContext, CrowdCL is a framework for distributing applications that can be easily embedded in any web page and aggregating the results of clients. A CrowdCL implementation of the Thomson problem significantly outperformed serial implementations in Javascript and Java.

While the WebCL browser extension remains under active development, our results show strong promise for cross-platform parallel computation on the web. As support for GPUs continues to improve across web browsers, we hope the ease of use of development tools for web-based GPU computations reflects the increasing viability of the browser as a platform for parallel computation.

REFERENCES

[1] CrowdCL implementation, May 2013. Available: http://github.com/tmacwill/crowdcl.

[2] WebCL KernelContext implementation, May 2013. Available: http://github.com/tmacwill/webcl-kernelcontext.

[3] E. Aho, K. Kuusilinna, T. Aarnio, J. Pietiainen, and J. Nikara. Towards real-time applications in mobile web browsers. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*, pages 57–66, 2012.

[4] E. L. Altschuler, A. Perez-Garrido, and R. Stong. A novel symmetric four dimensional polytope found using optimization strategies inspired by thomson's problem of charges on a sphere. 01 2006.

[5] D. Anderson. BOINC: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004.

[6] A. R. Bausch, M. J. Bowick, A. Cacciuto, A. D. Dinsmore, M. F. Hsu, D. R. Nelson, M. G. Nikolaides, A. Travesset, and D. A. Weitz. Grain boundary scars and spherical crystallography. *Science*, 299(5613):1716–1718, 2003.

[7] A. Beberg, D. Ensign, G. Jayachandran, S. Khaliq, and V. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009.

[8] C. Cecka, A. Middleton, and M. Bowick. Thomson problem database and applet, May 2013. Available: http://thomson.phy.syr.edu/.

[9] P. Delsarte, J. Goethals, and J. Seidel. Spherical codes and designs. *Geometriae Dedicata*, 6(3):363–388, 1977.

[10] T. Erber and G. M. Hockney. *Complex Systems: Equilibrium Configurations of N Equal Charges on a Sphere (2 N 112)*, pages 495–594. John Wiley & Sons, Inc., 2007.

[11] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 453–456, New York, NY, USA, 2008. ACM.

[12] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.

[13] H. W. Kroto, J. R. Heath, S. C. O'Brien, R. F. Curl, and R. E. Smalley. C60: Buckminsterfullerene. *Nature*, 318(6042):162–163, 11 1985.

[14] J. Lidmar, L. Mirny, and D. Nelson. Virus shapes and buckling transitions in spherical shells. *Phys Rev E Stat Nonlin Soft Matter Phys*, 68(5 Pt 1):051910, 2003.

[15] E. Rakhmanov, E. Saff, and Y. Zhou. Electrons on the sphere. *SERIES IN APPROXIMATIONS AND DECOMPOSITIONS*, 5:293–310, 1994.

[16] J. Thomson. Xxiv. on the structure of the atom: an investigation of the stability and periods of oscillation of a number of corpuscles arranged at equal intervals around the circumference of a circle; with application of the results to the theory of atomic structure. *Philosophical Magazine Series 6*, 7(39):237–265, 1904.