Just-in-time Acceleration of JavaScript

Uday Pitambare Arun Chauhan Saurabh Malviya

School of Informatics and Computing, Indiana University, Bloomington, IN 47405
{upitamba,achauhan,malviyas}@cs.indiana.edu

Technical Report TR706, February 2013

Abstract

JavaScript has seen tremendous growth in popularity driven by increasingly interactive web sites and sophisticated web interfaces. However, the performance of JavaScript continues to be a hurdle in using it for tasks that are computationally intensive, such as gaming, simulations, and visualization. JavaScript has also been slow to exploit the available parallelism on modern computers. Specifically, it is not currently easy to exploit GPGPUs within JavaScript. A part of the reason is that the low-level interface demanded for GPGPU programming is often not approachable by JavaScript programmers.

In this paper, we present a novel approach that provides a mechanism to accelerate portions of JavaScript programs without requiring the programmers to learn new syntax or low-level APIs. We achieve that through an embedded DSL used to specify GPGPU computations. We have designed a JavaScript library, and an accompanying Firefox extension, that work together to compile the embedded DSL just-in-time using the LLVM backend for generating PTX. The compiled code is cached to minimize the compilation overhead. Our evaluation of the system using a micro-benchmark, two applications kernels, and an application benchmark demonstrates that our approach imposes minimal performance overhead, while providing an easy GPGPU programming interface to JavaScript programmers.

1 Introduction

Web sites are increasingly dynamic today, with high levels of interactivity and real-time content. A large fraction of content is also consumed by a growing number of mobile devices on wireless networks, where bandwidth conservation is a high priority. As a result, web pages rely on client-side scripting to improve the user experience. Standardization of JavaScript has made it the language of choice for platform-independent and portable client-side scripting [3].

Building on the large body of work on just-in-time optimization of dynamic languages [1, 6], a considerable effort is being spent on optimizing the performance of JavaScript [23, 31, 28, 11]. The popularity of mobile platforms has also triggered optimization efforts directed specifically at saving energy [18]. More recently, there have been efforts at leveraging multiple CPU cores, such as Intel Labs RiverTrail [24] and some attempt at providing access to GPUs within JavaScript, such as WebCL [29]. While the former provides higher level abstractions to the programmers, it is not available on GPUs, and the latter provides a relatively low-level application programming interface (API) to OpenCL [8].

In this paper we describe an embedded domain-specific language (DSL) that we have designed and implemented for providing a high-level abstraction within JavaScript for writing snippets of code that could run on GPUs. The embedded DSL, called Harlan-J, is inspired by Harlan, which is a more general GPU programming language [10]. Harlan-J allows users to specify the computations to be accelerated on GPGPUs, within their JavaScript programs, using the familiar JavaScript syntax and without having to write boilerplate code for device initialization and data transfer. The use of embedded DSL also makes the approach portable—an important consideration for JavaScript programs—by hiding the platform-specific details within the library. The code could potentially also be compiled and optimized for multi-core CPUs if the execution environment does not provide a GPGPU.

We have implemented a prototype of the DSL through a JavaScript library and a Firefox extension that compiles the user code just-in-time (JIT) by using the PTX backend of LLVM [16, 22]. The translation is performed only the first time the code is encountered. Subsequent invocations make use of a cached copy, thus eliminating the compilation overheads. We present performance results on a micro-benchmark. two application kernel benchmarks, and an application benchmark to characterize our system. The empirical measurements demonstrate that the embedded DSL adds negligible overheads, resulting in potentially substantial gains in programmer's produc-

tivity without any significant performance penalty.

The main contributions of the paper include:

- Design of an embedded DSL for specifying GPU computations within JavaScript, using the standard language syntax with which programmers are already familiar.
- A prototype implementation of the DSL.
- Empirical evaluation of the implementation on two application benchmarks and a set of micro-benchmarks.

To the best of our knowledge, this is the first high-level JIT-compiled system to make GPU programming available within JavaScript as an embedded DSL, without encumbering the programmers with boilerplate code that relatively lower level programming environments require.

2 Related Work

JavaScript performance has attracted wide attention with its rise as the preferred language for cross-platform client-side scripting in browsers. The growing adoption of HTML5 may only increase the importance of JavaScript performance.

There are many compilation techniques, several of them just-in-time, that are geared toward dynamically typed languages [1, 6]. These techniques are also largely applicable to JavaScript. Leading implementations of JavaScript interpreters have started a race to achieve the best JavaScript performance in browsers [31, 28, 11]. Motivated by their extensive use on mobile platforms, there has also been interest in improving the energy performance of JavaScript [18, 26].

Recently, Richards et al., have analyzed JavaScript code in great detail [23]. They studied the dynamic behavior of JavaScript programs by identifying and categorizing a variety of characteristics. Based on scripts gleaned from 17 representative web-sites, one conclusion of the study was that the "execution time is dominated by hot loops", although less so than in Java. This observation is significant since the programs that were studied were drawn from a variety of sources, not just those restricted to specific domains, such as engineering or gaming, in which loops traditionally constitute the bulk of computations. The presence of hot sections in code provides a strong motivation to our approach that is based on accelerating some of those hot sections on GPUs. Further, the fact that hot sections are often loops makes that code more amenable to accelerating on GPUs.

An additional motivation behind our approach is the likelihood of JavaScript getting used for applications that would currently be impractically slow, if its execution could be dramatically improved. This likelihood stems from a wider adoption of JavaScript in application domains such as, gaming and image processing, driven by the attraction of portable applications that are always up to date and require no installation steps by the users, many of whom might be hesitant in installing third-party software on their machines.

It is possible for browsers and JavaScript interpreters to make use of the parallelism of multiple cores on a machine, and even GPUs for certain functions, such as rendering [14, 15]. However, providing parallelism within user-level scripts requires additional support. Intel's RiverTrail project lets JavaScript programs directly leverage the parallelism available on modern chip multiprocessors [24]. RiverTrail provides a high level data-parallel model for writing parallel code. However, it does not support GPUs.

WebGL and WebCL are the two leading systems that allow JavaScript programmers to use GPUs [30, 29]. WebGL provides JavaScript programmers the same model of programming that OpenGL does [19]. This includes abstractions such as canvas and shader. While it provides the full power of OpenGL to JavaScript programmers, it is not convenient for programming GPUs for general purpose computing. WebCL attempts to address that problem by providing an OpenCL-style interface within JavaScript. While this makes GPGPU programming significantly simpler than using WebGL, it is still too complex—and too low level—for casual JavaScript programmers who wish to accelerate portions of their code.

In summary, Harlan-J aims at raising the level of abstraction at which JavaScript programs leverage GPUs. We achieve this by involving a JIT compiler framework. To the best of our knowledge, no other similar system exists for JavaScript at the time of writing of this paper.

Harlan-J is inspired by Harlan, which is a high-level language that allows arbitrary expressions to be computed on GPUs [10]. It is designed as a general-purpose language and is compiled offline, ahead of run time. In contrast Harlan-J is an embedded DSL, which constrains it to follow the host language (JavaScript) syntax. Moreover, since it is compiled just-in-time, it may not rely on extensive analyses for compilation and optimization. Finally, the unique characteristics of JavaScript, especially related to security, pose unique challenges in implementing Harlan-J.

Several recent attempts have been made to increase the level of abstraction for programming GPUs, including translating OpenMP to GPUs [17], automatic generation of GPU code from affine loops [2], declarative models for programming GPUs [7], C++ template libraries of algorithms that can be compiled for GPUs [9], and automatically leveraging GPUs from within high-level languages, such as MATLAB [25]. However, none of these efforts has

targeted the JavaScript language.

3 Language Design

Our goal is to enable JavaScript programmers to accelerate sections of their code at a higher level of abstraction than afforded by currently available lower level APIs. At the same time we would like relatively advanced programmers to be able to tune the performance of their code for specific GPUs or GPU architectures. The goal is to design an abstraction that hides platform-specific details and provides a uniform interface to leverage data parallelism on a variety of parallel platforms, including (possibly) chip-multiprocessors.

With this in mind, we have followed three design principles:

1. *Familiar syntax for common case*
   We chose to design Harlan-J as an embedded DSL so that the user is unencumbered by the need to learn new syntax. This results in, potentially, higher productivity and more easily maintained code. In the common case, no special annotations are required. Syntactic sugar for commonly used variables and computations (such as computing the global thread ID) keeps the syntax for the common cases clean.

2. *Optimization features for advanced users*
   A recurring issue with higher level abstractions is the *abstraction penalty* that deters advanced users from using them. This is unfortunate, since abstractions make the code more readable, more portable, and better suited to automatic optimizations that could be retargeted for a new platform. In order to allow advanced programmers to tune the performance of their code we provide a small set of keywords that allow platform-specific optimizations, such as concurrency across kernels and allocations in specific memory types.

3. *Opportunities for automatic optimizations*
   An important design goal of Harlan-J has been to provide clear opportunities for the accompanying JIT compiler to perform optimizations. Since the optimizations must be simple enough to be suitable for JIT compilation, the opportunities must be discernible without extensive analysis.

3.1 Syntax

Figure 1 shows two examples written in Harlan-J. CUDA code for vector addition is shown to demonstrate the absence of boilerplate code in the Harlan-J version. The `HJ` namespace contains all the Harlan-J-related functions and variables.

Before a GPU code can be executed it must be *set up* through the `HJ.setupKernel` method, which takes a function representing the kernel. The setting up includes JIT-compiling the kernel. Once the kernel is compiled it may be executed with the `execute` method. The execution does not have to follow the set up immediately, and it may be invoked any number of times with varying actual parameters. Moreover, the order of setting up kernels need not be the same as executing them. However, when certain user-specified optimizations are used, the user must ensure that execution order does not violate data dependencies, as discussed below in Section 3.3.

All code outside the two methods, `HJ.setupKernel` and `HJ.execute`, is standard JavaScript. Only two types of arguments may be passed to the kernel, scalars or *typed arrays*. JavaScript language has only recently incorporated typed arrays that are designed specifically for interoperability with native binary data [27]. As a result, exchanging data with native code, written, say, in C, is highly efficient through typed arrays.

As the name suggests, typed arrays have specific types associated with them. As the examples show, first a raw buffer must be allocated and then a typed *view* must be associated with it—the buffer may not be manipulated directly. It is this view that is passed to the kernel, from which information about size and type of the array may be deduced. Typed arrays also support subarrays, which allows extracting regular sections from arrays.

Internally, Harlan-J supports all the basic types supported by JavaScript typed arrays. Table 1 lists these basic types and the typed array corresponding to each.

The kernel code is written in single instruction multiple data (SIMD) form, which could be expressed either by specifying operations on each data element of arrays passed as arguments, or by indexing into the arrays explicitly. The vector addition kernel is an example of the first and the matrix-multiply kernel illustrates a mixed strategy.

Inside the kernel several special variables and methods

Harlan-J type	*JavaScript typed array*
`float64`	`Flota64Array`
`float32`	`Float32Array`
`uint32`	`Uint32Array`
`int32`	`Int32Array`
`uint16`	`Uint16Array`
`int16`	`Int16Array`
`uint8`	`Uint8Array`
`int8`	`Int8Array`

Table 1: Basic types supported by Harlan-J and the corresponding JavaScript typed arrays.

CUDA code for adding two vectors

```
__global__ void add_kernel(int size, float *X, float *Y, float *Z)
{
  int i = threadIdx.x;
  if(i < size) { Z[i] = X[i] + Y[i]; }
}

void vector_add(int size, float *X, float *Y, float *Z)
{
  float *dX, *dY, *dZ;
  cudaMalloc(&dX, size * sizeof(float));
  cudaMalloc(&dY, size * sizeof(float));
  cudaMalloc(&dZ, size * sizeof(float));

  cudaMemcpy(dX, X, size * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(dY, Y, size * sizeof(float), cudaMemcpyHostToDevice);

  add_kernel<<<1, size>>>(size, dX, dY, dZ);

  cudaMemcpy(Z, dZ, size * sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree(dX);
  cudaFree(dY);
  cudaFree(dZ);
}
```

Harlan-J code for adding two vectors

```
// Declare and initialize typed one-dimensional arrays A, B, and C
var A_buf = new ArrayBuffer(N*8); // for N float64
var A = new Float64Array(A_buf);  // create a view
...
var vec_add = HJ.setupKernel(
   function (A, B, C)
   {
     HJ.element({a: A, b: B, c: C});
     c = a + b;
   });
vec_add.execute(A, B, C);
```

Harlan-J code for matrix-matrix multiply

```
// Declare and initialize typed matrices A, B, and C
var A_buf = new ArrayBuffer(N*N*8); // for N*N float64
var A = new Float64Array(A_buf);    // create a view
...
var matmul = HJ.setupKernel(
   function (A, B, C, N)
   {
     HJ.element({C: c});
     for (var i=0; i < N; i++) {
       c += A[HJ.myID*N+i] * B[i*N+HJ.myID];
     }
   }
matmul.execute(A, B, C, N);
```


Figure 1: CUDA code for adding two vectors, followed by `harlanj` codes for vector addition and matrix-matrix

are made available by Harlan-J.

- `HJ.myID` is the global thread ID of the current thread.
- `HJ.blockDim.x`, `HJ.blockID.x`, and `HJ.threadID.x` correspond to the number of blocks, the block number, and the offset of the current thread within the block, respectively, along the `x` dimension. Similar variables exist for `y` and `z` dimensions.
- `HJ.element` method takes an object map and uses it to generate variables that contain elements corresponding to `HJ.myID`.
- `HJ.neighbor` method takes an *offset* to generate an index relative to the current thread. The offset could be multidimensional (multiple values, one for each dimension) for multidimensional thread layouts.

In Figure 1 the vector addition kernel accepts three arrays, `A`, `B`, and `C`. The `HJ.element` method is used to access an element of each array to express the SIMD addition in the next statement. The programmer does not explicitly specify input and output data, which the compiler determines by doing a def-use analysis. However, the programmer may explicitly specify what (not) to copy for optimizing performance, as described in Section 3.3.

In the matrix-multiply kernel only the array `C` is accessed element-wise. The kernel makes use of the convenience variable, `HJ.myID`, to access the multiple elements from the other two arrays. The computation may be improved by parallelizing the loop inside the kernel as a reduction. This could be done manually, but a more compelling alternative would be to abstract that away in a *reduction kernel* that is nested inside this kernel. Our current implementation does not support reduction or nested kernels, but Section 7 discusses this possibility.

By default, the compiler generates a one-dimensional sequence of blocks within the grid and a one-dimensional sequence of threads in each block. However, it may be overridden by explicitly specifying multi-dimensional allocation, as discussed in Section 3.3. A default number of threads is picked, based on the recommended number for the GPU, and the default number of blocks is computed based on the array size and the number of blocks. However, these can be overridden by passing additional arguments to `execute`.

3.2 Semantic Limits

We have made the design choice of restricting the permissible operations and data structures within kernel code. This design choice is driven partly by the impracticality of supporting a large number of operations and arbitrary JavaScript data structures in our compiler. However, another important motivation is to encourage programmers to be aware of the limits of the GPU hardware and pay attention to structuring their programs accordingly. For instance, arbitrary object creation or advanced control flow structures, such as closures, are not directly supported on current GPUs, which makes them highly inefficient to realize on GPUs. Similarly, operations on sparse or irregular data structures do not lend themselves easily to efficient SIMD execution, requiring algorithmic innovations to leverage the GPUs effectively.

A strategy to work around this limitation is to identify regular patterns in the computations that are amenable to efficient SIMD execution. Often this is possible to do even for algorithms that, at first sight, appear highly irregular, such as, compression [20]. By not hiding the limitations of the underlying GPUs, Harlan-J provides an incentive to the programmers to rework their application so that only the appropriate portions of the code are targeted for acceleration with GPUs which, we believe, is a more desirable approach in the long term compared to abstractions that might mislead programmers into writing highly inefficient GPU code.

Finally, by restricting the semantics of Harlan-J kernel operations and providing direct mapping of those operations to the underlying GPUs, it may be possible to perform quick optimizations that would be practical in a JIT compiler. More extensive semantics would require a deeper analysis, which is likely to be impractical in a JIT compiler.

3.3 User Optimizations

Harlan-J allows several optimizations to be specified as compiler directives. Figure 2 illustrates this with a dot-product kernel that is called repeatedly in a loop.

Preserving data across kernel invocations A sequence of kernel calls might operate on a common set of data or a kernel might produce data needed by an kernel that follows immediately. If the data items are read-only, or if they are not needed in code between successive calls to the kernel, then it is possible to avoid multiple data transfers. The `dotProd` kernel in Figure 2 preserves array `B` since it does not change across kernel invocations. Note that this information is impossible for the compiler to deduce since it does not analyze any code outside the kernel.

Always-copy or never-copy The compiler can also be overly conservative in determining the output variables, since it does not have an accurate set of live variables at the end of the kernel. In such cases, it may be useful for the programmer to explicitly forbid the compiler from

Repeated dot-product with one operand preserved across kernel invocations

```
// Allocation and initialization
// of variables elided

var dotProd = HJ.setupKernel(
    function (A, B, D)
    {
      HJ.preserve(B);
      HJ.element({a: A, b: B, d: D})
      d = a * b;
    });
var dot = 0.0;
for (var i=0; i < N; i++) {
  dotProd.execute(A, B, D);
  for (var j=0; j < block; j++)
    dot += D[j];
  some_computation(dot);
  update_value(A);
}
```

Explicitly copying required data, but not other

```
...
var dotProd = HJ.setupKernel(
    function (A, B, D)
    {
      HJ.preserve(B);
      HJ.alwaysCopyOut(D);
      HJ.neverCopyOut(A, B);
      HJ.element({a: A, b: B. d: D})
      d = a * b;
    });
...
```

Making the kernel call asynchronous

```
...
for (var i=0; i < N; i++) {
  var k = dotProd.spawn(A, B, D);
  update_value(A);
  HJ.wait(k);
  for (var j=0; j < block; j++)
    dot += D[j];
  some_computation(dot);
}

...
```

Figure 2: An example of repeated dot-product illustrating some of the user optimizations supported in Harlan-J.

generating copy-out code for some variables, by using the `HJ.neverCopyOut` method.

On the other hand, the user might also want to force copying out of certain data, perhaps, for debugging. The method `HJ.alwaysCopyOut` directs the compiler to insert copy out code for specified variables irrespective of whether the variables are written in the kernel.

Asynchronous kernels While GPUs commonly support asynchronous kernel calls we have chosen to not make that the default behavior in order to keep the kernel semantics simple. However, a kernel may be called asynchronously by using the `spawn` method, instead of `execute`. In that case a handle returned by `spawn` may then be used to `wait` for the kernel when its output is needed. The running example in Figure 2 shows this in the third code segment, overlapping the kernel execution with computing the array `A` for the next iteration.

Further overlapping is possible by unrolling the loop, which is not shown. Unrolling would make it possible to overlap the kernel for the next iteration with computing `dot` and `some_computation` with `dot` in the current iteration.

Synchronizing threads A primitive to synchronizing threads is not strictly necessary, since there is an implicit barrier at the and of each kernel. Therefore, any thread synchronization within the kernel may be realized by splitting the kernel into multiple kernels. Therefore, we treat the synchronization primitive as an optimization. `HJ.sync_block` is equivalent to CUDA `__sync_threads`.

Controlling memory allocation Harlan-J lets users specify if certain variables should be allocated in *global*, *shared*, *constant*, or *texture* memory—the default is *global*. This is done with four data placement directives `HJ.placeGlobal`, `HJ.placeShared`, `HJ.placeConstant`, and `HJ.placeTexture`, respectively. The compiler automatically generates code to transfer data back-and-forth from global memory, as needed. However, the user is responsible for using the memory regions correctly, for example not attempting to write into constant memory.

Multi-dimensional blocks and threads Multi-dimensional blocks or threads may be specified with `HJ.threadDims` and `HJ.blockDims` primitives. The default is one dimensional blocks and one dimensional distribution of threads.

4 Language Implementation

We have implemented a prototype of Harlan-J for Mozilla Firefox. This section describes the challenges faced in

implementing the JIT-compiler and our solutions. We use LLVM's [16] PTX back-end [22] to translate Harlan-J kernels into PTX and Mozilla's `js-ctypes` library [13] to interface with a C library that can communicate directly with the GPU.

It is important to note that our approach assumes that the JavaScript program runs in the privileged mode. This makes Harlan-J in its current form suitable for writing extensions, but unsuitable for untrusted content scripts loaded through the browser. There are mechanisms that make it possible to communicate data back and forth between privileged and untrusted scripts, for example, through DOM events [12]. The content script as well as the privileged code (say, an extension) register DOM event listeners that are triggered when new data arrive within a `div`. Two-way communication can happen by adding the data to the appropriate DOM elements to which event listeners are attached. Another possible mechanism in Firefox is through `XPConnect` wrappers [32]. However, we have not explored those options to interface untrusted JavaScript code to access the GPU. Both these options seem likely to have high overheads due to the enforcement of security policies related to interactions between untrusted and privileged JavaScript code.

Figure 3 illustrates the overall system components. The Harlan-J framework implements four main steps:

1. A JavaScript library, called `esprima` [5], is used to generate a JavaScript abstract syntax tree (AST) from arbitrary JavaScript code passed in as a string. The built-in JavaScript unparsing method, called `toString`, is used to convert the function passed to `HJ.setupKernel` into a string.

2. The AST is of the Harlan-J kernel is translated directly to LLVM intermediate representation. This makes use of the `escodegen` library [4] to modify the AST and LLVM APIs to generate code.

3. The `js-ctype` library is used to invoke the LLVM backend to convert the intermediate representation to PTX using the LLVM's PTX backend.

4. Finally, a wrapper object is returned to the user program that can be subsequently used to invoke the PTX kernel.

4.1 Firefox Extension

The JavaScript code using Harlan needs to to run in privileged mode to invoke system services that would otherwise not be accessible to untrusted JavaScript. We use the Mozilla `js-ctypes` library to load and invoke system shared libraries [13]. This mechanism is used to call the LLVM backend and to access the GPU services.

While this limits Harlan-J to code within extensions, there are mechanisms that can be used to establish communication between untrusted content scripts and privileged code, as discussed earlier in this section. However, we have not measured their overheads and effectiveness to serve as a conduit to offload computations on to the GPUs.

4.2 JIT Translator

The just-in-time translator is a major component of the Harlan-J framework. It walks through the kernel AST, and generates the LLVM IR. Algorithm 1 shows the core code generation algorithm. The algorithm outlines a simplified version of our implementation, omitting several details regarding generating GPU boilerplate code, data copying primitives, variable declarations, and argument passing.

The compiler analyzes the code for use of any special variables, as described in Section 3, and generates additional code to initialize those variables within the kernel.

4.3 Handling Harlan-J Primitives

Harlan-J primitives are processed before traversing the AST to generate LLVM intermediate representation by Algorithm 1. We discuss each primitive below, most of which concern optimizations. The primitives are processed in a flow-insensitive manner, meaning that their impact is *global*, throughout the kernel. The compiler first makes a pass through the kernel AST to collect all the primitives and record their impact, which guides the rest of code generation.

HJ.element The `HJ.element` primitive triggers creation of new variables, based on input array arguments, as specified by the hash-map passed to the primitive. The compiler uses the global thread ID to generate assignments from the appropriate array indexes into the corresponding scalar element variable. The type of the element can be deduced from the type of the array.

HJ.preserve By default, global memory on the device is preserved across kernel calls. Therefore, if a variable that needs to be preserved is allocated in the global memory all that the compiler needs to ensure is to not generate code to free that variable when the kernel ends. If the variable to be preserved is in another region of memory, say shared memory, then the compiler generates code to copy it to a global location. Note that the subsequent calls to the kernel must know the variable location and if the subsequent kernel is unrelated (say, in another stream) it will need to be passed a pointer to the variable.

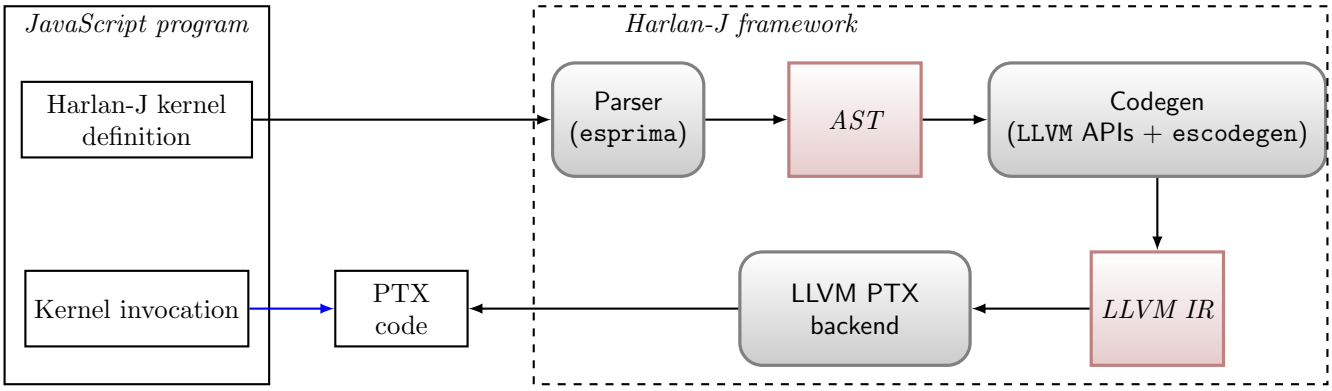

Figure 3: Overview of the Harlan-J system components.

Algorithm: CODEGEN
Input: Harlan-J kernel AST, T
Output: LLVM IR, R

---

if T *is of the form* `x = e`, *where* `e` *is an expression* **then**
  $R \leftarrow$ GENERATE(ASSIGN, CODEGEN(`x`), CODEGEN(`e`))
else if T *is of the form* `x binop y` **then**
  $R \leftarrow$ GENERATE(BINOP, CODEGEN(`x`), TRANSLATE(`binop`), CODEGEN(`b`))
else if T *is of the form* `unaryop x` **then**
  $R \leftarrow$ GENERATE(UNARYOP, TRANSLATE(`unaryop`), CODEGEN(`x`))
else if T *is of the form* `a`, *where* `a` *is a variable* **then**
  $R \leftarrow$ TRANSLATE(`a`)
else if T *is of the form* `if (e) S1 else S2` **then**
  $R \leftarrow$ GENERATE(IF, CODEGEN(`e`), CODEGEN(`S1`), CODEGEN(S2))
else if T *is of the form* `for (C1; C2; C3) S` **then**
  $R \leftarrow$ GENERATE(FOR, CODEGEN(`C1`), CODEGEN(`C2`), CODEGEN(`C3`), CODEGEN(`S`)
else if T *is a list of statements* **then**
  $R \leftarrow \varnothing$
  for *each statement* s *in* T **do**
    $R \leftarrow R \cup$ CODEGEN(s)
else
  ERROR("Unsupported statement type")
return R

Algorithm 1: The core algorithm to generate code works by a recursively descending down the AST. The helper function GENERATE generates the LLVM IR and TRANSLATE converts a variable name or operator to its LLVM equivalent. Some cases have been elided for the sake of clarity, for example, array subscripts.

HJ.alwaysCopyOut and HJ.neverCopyOut As the names suggest, the compiler ensures that variables specified by `alwaysCopyOut` are always copied back to the CPU memory and those specified by `neverCopyOut` are never copied back.

Asynchronous kernel execution This is really an issue of appropriate run time support. The `spawn` method does not wait for the kernel to finish, but instead returns immediately without copying the results back to the CPU. A later `wait` call synchronizes with the kernels and copies the data back to the CPU. The run time systems maintains pointer associations between GPU and CPU locations to enable the data copying.

Synchronizing threads The `HJ.sync_block` maps directly to the CUDA function `__syncthreads`.

Memory allocation The compiler honors all placement requests for variables by generating appropriate CUDA allocation blocks for LLVM's PTX back-end. However, the user is responsible for ensuring that the capacity of the memory regions are not exceeded and that they are used correctly, e.g., constant memory region is never modified.

Multi-dimensional blocks and threads Unlike CUDA, the dimensionality is specified within the kernel. Given that kernels are usually written with the assumption of a certain block and thread distribution, this seems a reasonable approach. Note that the number of threads and blocks can be specified with the `execute` or `spawn` methods. In order to implement this with CUDA, the compiler records the user specified dimensions in the kernel object that is returned to the user, which are used when the `execute` or `spawn` method is called.

4.4 Harlan-J as a Library

While we have described the Harlan-J framework in the context of a JIT compiler, our embedded DSL design also enables an implementation of Harlan-J as a pure library. This could be particularly useful when the underlying system has no GPU support.

Note that an absence of a GPU target does not automatically preclude a JIT compiler, since the kernel could still be compiled to multi-threaded native code to run on multiple cores of the CPU. Indeed, that is one of the motivations behind developing a higher level abstraction for specifying SIMD parallelism. However, if translation to native code is not desirable, or impossible due to security concerns, then a library-based implementation can be used to still leverage the data parallelism implied in a Harlan-J kernel.

In a library-based implementation, the kernel set up does not compile the kernel, but instead simply records the kernel code, which becomes a closure, in a wrapper object and returns the wrapper. The kernel `execute` method simply invokes the saved closure with the appropriate formal parameters. In such a case, the kernel is no different than standard JavaScript code.

However, the implementation can improve the performance of the kernel if it is allowed to create multiple threads. Since the kernel specifies a SIMD execution, each SIMD operation can be thought of as a *virtual thread*, which is mapped to a system thread. Note that a virtual thread does not need to be executed as a separate unit. It can be emulated simply by using an appropriate range of virtual block and thread IDs and a loop over the range within each system thread.

The dynamic features of JavaScript can be used to define element-wise variables declared through `HJ.element` primitive. Similarly, the primitives referring to the block and thread IDs and numbers map simply to the equivalent underlying CUDA variables. The `HJ.neighbor` primitive is easily implemented by computing the equivalent value based on the current thread ID.

If the "kernel" is going to be executed on the CPU then the GPU-specific user optimizations become null operations.

5 Experimental Evaluation

We implemented a prototype Harlan-J system on Firefox, using the PTX back-end of LLVM. We evaluated our prototype implementation using a microbenchmark, two applications kernels, and one application benchmark. The experiment was performed with Firefox version 17, LLVM 3.1 on a Tesla C1060 card. Since we needed a desktop environment to run the test within a browser, it precluded us from using newer generations of GPGPU cards that were only available to us in server environments.

5.1 Microbenchmarking

We wrote a microbenchmark to evaluate the overheads of compiling and initializing the device using our prototype Harlan-J implementation. Table 2 summarizes our findings. The first measurement was the time taken by the JIT compilation framework to compile a trivial kernel. This allowed us to estimate the overhead of the compiler, which ranged from 38 to 46 ms. The overhead does not change significantly for kernel sizes of up to a few hundred statements, which is the normal range of most kernel code (and is likely to be smaller for Harlan-J kernels without any boilerplate code).

A second measurement computed the time taken to initialize the device, which is about 12 ms. Once again, it

Operation	*Time range (ms)*
System initialization	12–13
Set up time	38–46

Table 2: JIT compiler overheads.

is a one-time cost, paid only once per *script*. As a result, we consider it a manageable cost.

While the compilation overhead seems high at first, it needs to be paid only once per kernel. Moreover, similar to JavaScript caching techniques, the compiled kernels could also be cached, thus reducing the overheads greatly. Another potential to reduce the overheads arises out of an artifact of our current implementation, which generates LLVM assembly and then invokes the LLVM assembler to convert it into the LLVM "bit-code." Generating the LLVM bit-code directly from Harlan-J kernel could reduce the translation overheads. Finally, it is also possible to generate PTX directly. However, in that case, we would lose the benefits of LLVM's built-in optimizations and the portability gained by using the LLVM back-end.

5.2 Application Kernels and Benchmarks

Next, we evaluated our system on two application kernels and one application benchmark. The application kernels are vector addition and dense matrix-multiply. The application benchmark we used was N-Body simulation, which we tested in two different versions. The first version uses a single force calculation step and another that performs 1000 steps in rapid succession to stress-test our dispatch system. Figure 5 plots the results for the four experiments with varying input sizes. For vector addition, the input size is the number of vector elements, for matrix multiply it is the dimension of a square matrix, and for N-Body simulation it is the number of bodies.

In each case, we compared the execution time of the kernel compiled through Harlan-J framework and called from within JavaScript with one generated from CUDA and called from within a C program. Note that we did not compare with pure JavaScript, because that is orders of magnitude slower. It is clear that the GPU version of the code would be much faster than sequential JavaScript. Our goal was to evaluate if we could match the performance of PTX kernel generated from CUDA and called from C. Note that we include the data transfer time in the total execution time.

For both vector addition and matrix multiplication Harlan-J performance closely follows that of CUDA-generated kernel. In the case of vector addition, it is sometimes even slightly faster, but we attribute that small difference to some amount of noise in measurements.

We then evaluated the our prototype on N-Body force calculation simulation, adapted[1] from its original OpenCL form, designed to stress test our system. Figure 4 outlines the code with the kernel called inside a loop.

```
// Variable declarations and initialization
...
var nbody = HJ.setupKernel(
function (fpx, fpy, fpz,
          fpxnew, fpynew, fpznew,
          fpvx, fpvy, fpvz, fpm) {
    // variable declarations
      ...
    for(ij=0; ij<k; ij++) {
      fdx = fpx[ij] - fpx[iglobalId];
      fdy = fpy[ij] - fpy[iglobalId];
      fdz = fpz[ij] - fpz[iglobalId];
      fdistSqrt = 1.0/sqrt(fdx * fdx +
                           fdy * fdy +
                           fdz * fdz +
                           feps);
      fdistSixth = fdistSqrt *
                   fdistSqrt *
                   fdistSqrt;
      ff = fpm[ij] * fdistSixth;
      fax += ff * fdx;
      fay += ff * fdy;
      faz += ff * fdz;
    }
    // writing results
      ...
});
...

for (var i=0; i < N; i++) {
  nbody.execute(float32ViewX.buffer,
                float32ViewY.buffer,
                float32ViewZ.buffer,
                float32ViewXnew.buffer,
                float32ViewYnew.buffer,
                float32ViewZnew.buffer,
                float32ViewVX.buffer,
                float32ViewVY.buffer,
                float32ViewVZ.buffer,
                float32ViewM.buffer);
  // update
  ...
}
```

Figure 4: Outline of the iterative N-Body kernel designed to stress-test the Harlan-J execution system.

[1] Adapted from: `http://www.browndeertechnology.com/docs/`

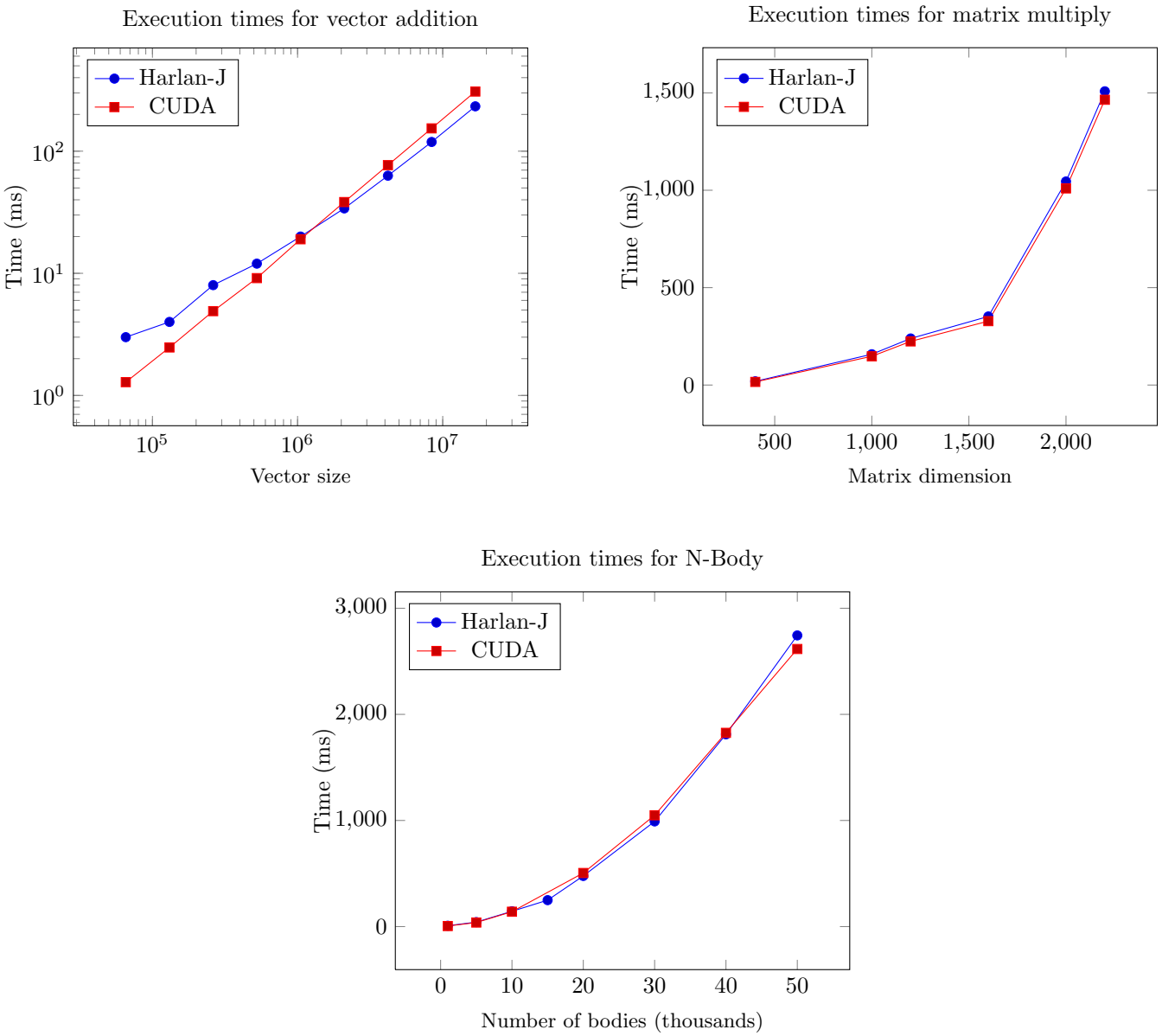

Figure 5: Comparing the execution time of PTX generated from CUDA and executed from within a C program, to the PTX generated by the Harlan-J framework and executed from within JavaScript. Note that these times do not include the code generation time, but do include data transfer times between CPU and GPU.

The way this code stresses our system is by its structure. It contains a non-trivial amount of computation in the kernel and outside, several local variable declarations within the kernel, a long list of arguments, and control flow constructs within the kernel. It tests the ability of our system to generate complex code. The bottom left plot in Figure 5 shows the comparison of the running time of this kernel with an equivalent CUDA-generated kernel called from C, for a single step. The running times are practically identical.

In summary, the Harlan-J system imposes a high overhead in the initial compilation phase, but there is significant potential for improvement. However, once a kernel has been compiled the overhead of actually invoking it from JavaScript is negligible, compared to C.

6 Security

Security is an important issue in letting untrusted third-party code execute on local machine. Browsers implement strict policies to limit resource access by untrusted JavaScript. Providing such code access to a machine's GPU might seem like opening a security hole. Even though we have not directly addressed the security issue in this paper, we make several observations here related to potential threats.

Denial-of-service attack A malicious or buggy program could make the GPU unavailable to any other program. In principle, this is not different than a runaway script. However, security and privacy issues with GPUs will need further investigation and might require additional support from the libraries that interface with the GPU.

Indirect access to other resources Another potential problem is a script gaining access to resources that it would normally not be allowed to access. This could happen by exploiting vulnerabilities in systems unrelated to the browser or JavaScript, but related to the GPU subsystem.

Leaking information Since the focus of GPU computing has been performance, several aspects will need to be looked at afresh if untrusted JavaScript is to be allowed access to them. For instance, values left around in memory by earlier programs could be read by malicious scripts and communicated to a third party, in order to glean any useful information. It could also allow multiple scripts, with different origins, using the GPU to communicate with each other—something that is prohibited by the JavaScript security model. Some of the issues here might be similar to those occurring in cloud-computing scenarios [21].

Cost of security The overheads caused by enforcing security and privacy policies will need to be included when the performance gains from GPUs are estimated. The traditional models of performance will have to be tweaked to make decisions regarding when and what to accelerate with the GPUs.

7 Future Work

The work opens several directions for future work.

Security Security issues need to be more completely studied and addressed in order to extend this facility to content scripts. The issues include protecting against denial-of-service by malicious scripts, privacy violation through information leak and other covert channels, and possibility of exploiting vulnerabilities not directly related to the browser or JavaScript interpreter. These will need careful considerations before the GPUs could be exposed to untrusted JavaScript.

Reducing compilation overheads The prototype generates LLVM assembly and then compiles it, which introduces an extra layer. It could generate LLVM bitcode or even be hooked directly with the LLVM back-end to generate PTX directly. This could reduce overheads, but further study will need to be undertaken to quantify the benefits.

Nested kernels Latest generation GPU cards allow nesting kernels, which could be directly supported in Harlan-J. Another related and important capability would be native support for parallel reduction, which will could greatly improve the expressiveness of the language for a wider class of algorithms.

Multidimensional arrays and other syntactic sugar Since array-based computation is an important class of SIMD applications, multidimensional array syntax with support for array sections might be an important syntactic sugar to support. However, a challenge here would be to find a clean syntax that could be embedded within the JavaScript language.

Increased concurrency Modern GPUs allow multiple concurrent kernels. They also allow concurrent data transfers. This could be an important optimization to

`BDT_OpenCL_Tutorial_NBody-rev3.html`

exploit, perhaps, through additional annotations. Asynchronous kernels leverage this concurrency to some extent. But, more precise annotations could help advanced users write more efficient code.

Extend to OpenCL and other platforms The current prototype works with a specific browser and specific hardware platform. It would be useful to be able to extend it to other back-ends, potentially including CPUs, and other browser platforms.

8 Conclusion

We have demonstrated the feasibility of a high-level system for programming GPGPUs within JavaScript. We did this by developing a high-level DSL called Harlan-J, embedded within JavaScript. Embedding within a highly dynamic language throws many challenges in defining the syntax and semantics of the language and implementing a JIT compiler. Harlan-J has been designed to make the common case easy, while providing advanced users adequate handles to be able to tune the performance of their GPU code.

We implemented a prototype of the language for Firefox browser and NVIDIA hardware. We used the `esprima` JavaScript library to parse the Harlan-J kernel code. The resulting AST is translated into LLVM intermediate representation and LLVM's PTX back-end is then used through the LLVM API to generate PTX code. Once the kernel is translated it may be called multiple times. Our experiments show that while the initial compilation overhead could be significant, it could be reduced through caching and improvements in the compiler's workflow. More importantly, our experiments demonstrate that the *execution* overhead of these kernels is small and comparable to calling an equivalent PTX kernel generated from CUDA and called from within C. This makes our approach an attractive way to make GPUs available to JavaScript programmers.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005. DOI: 10.1109/JPROC.2004.840305.

[2] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *In Proceedings of the 19th International Conference on Compiler Construction (CC)*, pages 244–263, 2010. DOI: 10.1007/978-3-642-11970-5.

[3] S. ECMA-262. ECMAScript language specification, edition 5.1. Technical Report ISO/IEC 16262:2011, ECMA International, June 2011.

[4] ECMAScript code generator. `https://github.com/Constellation/escodegen`.

[5] ECMAScript parsing infrastructure for multipurpose analysis. `http://esprima.org/`.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings o the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 465–478, 2009. DOI: 10.1145/1542476.1542528.

[7] M. Grossman, A. S. Sbîrlea, Z. Budimlić, and V. Sarkar. CnC-CUDA: Declarative programming for GPUs. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, pages 230–245, 2010. DOI: citation.cfm?id=1964536.1964552.

[8] K. Group. OpenCL: The open standard for parallel programming of heterogeneous systems. On the web. `http://www.khronos.org/opencl/`.

[9] J. Hoberock and N. Bell. Thrust – parallel algorithms library. `http://thrust.github.com`.

[10] E. Holk, W. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for GPUs. In K. D. Bosschere, E. H. D'hollander, G. R. Joubert, D. Padua, F. Peters, and M. Sawyer, editors, *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 297–304. IOS Press, Amsterdam, Netherlands, 2012. Proceedings of the 14th biennial ParCo Conference, 2011. DOI: 10.3233/978-1-61499-041-3-297.

[11] Mozilla IonMonkey. `http://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/`.

[12] Interaction between privileged and non-privileged pages. `https://developer.mozilla.org/en-US/docs/Code_snippets/Interaction_between_privileged_and_non-privileged_pages`.

[13] Js-ctypes. `https://developer.mozilla.org/en-US/docs/Mozilla/js-ctypes`.

[14] K. Kerr. Introducing Direct2D. http://msdn.microsoft.com/en-us/magazine/dd861344.aspx.

[15] V. Kokkevis. GPU accelerated compositing in Chrome. http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[17] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009. DOI: 10.1145/1504176.1504194.

[18] S.-W. Lee and S.-M. Moon. Selective just-in-time compilation of client-side mobile JavaScript engine. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011. DOI: 10.1145/2038698.2038703.

[19] OpenGL: The industry's foundation of high performance graphics. http://www.opengl.org.

[20] A. Ozsoy, M. Swany, and A. Chauhan. Pipelined parallel LZSS for streaming data compression on GPGPUs. In *Proceedings of the 18h IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2012. To appear.

[21] S. Pearson and G. Yee, editors. *Privacy and Security for Cloud Computing*. Computer Communications and Networks. Springer, 2013. DOI: 10.1007/978-1-4471-4189-1.

[22] H. Rhodin. *A PTX Code Generator for LLVM*. Bachelors thesis, Saarland University, Oct. 2010.

[23] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010. DOI: 10.1145/1806596.1806598.

[24] Intel Labs RiverTrail project. https://github.com/RiverTrail/RiverTrail.

[25] C.-Y. Shei, P. Ratnalikar, and A. Chauhan. Automating GPU computing in MATLAB. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 245–254, 2011. DOI: 10.1145/1995896.1995936.

[26] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who killed my battery?: Analyzing mobile browser energy consumption. In *Proceedings of the 21st International Conference on World Wide Web (WWW)*, pages 41–50, 2012. DOI: 10.1145/2187836.2187843.

[27] Typed array specification. http://www.khronos.org/registry/typedarray/specs/latest/.

[28] V8 JavaScript engine. http://code.google.com/p/v8/.

[29] WebCL – heterogeneous parallel computing in HTML5 web browsers. http://www.khronos.org/webcl/.

[30] WebGL. http://www.khronos.org/webgl/.

[31] The WebKit open source project. http://www.webkit.org.

[32] XPConnect wrappers. https://developer.mozilla.org/en-US/docs/XPConnect_wrappers.