ParallelJS: An Execution Framework for JavaScript on Heterogeneous Systems

Jin Wang
Georgia Institute of Technology
Atlanta, 30332
jin.wang@gatech.edu

Norman Rubin
NVIDIA Research
California, 95050
nrubin@nvidia.com

Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, 30332
sudha@ece.gatech.edu

JavaScript has been recognized as one of the most widely used script languages. Optimizations of JavaScript engines on mainstream web browsers enable efficient execution of JavaScript programs on CPUs. However, running JavaScript applications on emerging heterogeneous architectures that feature massively parallel hardware such as GPUs has not been well studied.

This paper proposes a framework for flexible mapping of JavaScript onto heterogeneous systems that have both CPUs and GPUs. The framework includes a frontend compiler, a construct library and a runtime system. JavaScript programs written with high-level constructs are compiled to GPU binary code and scheduled to GPUs by the runtime. Experiments show that the proposed framework achieves up to **26.8x** speedup executing JavaScript applications on parallel GPUs over a mainstream web browser that runs on CPUs.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; D.3.2 [**Programming Languages**]: Concurrent, distributed, and parallel languages

General Terms

Languages, Performance

Keywords

GPGPU, JavaScript, Parallel Construct

1. INTRODUCTION

The development of modern software is accompanied by the advance of high-level programming languages and environments, which emerges to address productivity and algorithmic issues. Simultaneously, the community of high performance computing are in a growing demand for the ability to harness massively parallel hardware such as Graphic Processing Units (GPUs). The gap between programming languages and environments that designed for productivity, and heterogeneous systems optimized for massive parallelism, speed, and energy efficiency introduces challenges to the compiler techniques, application design and runtime systems.

As one of the most prevalent script language, JavaScript has been well accepted for its high productivity and portability. JavaScript runs on essentially all platforms, both client and servers. It is also a deterministic language, which requires a single thread of control and heavy use of events. While recent JavaScript implementations have greatly improved performance, developers are still often forced to limit web content because JavaScript still does not take advantage of the massive parallelism available on modern hardware.

This paper proposes a framework, ParallelJS, which enables developers to use high performance data parallel accelerators such as GPUs in standard web applications. ParallelJS provides a simple language extension that offers massive speedup over sequential JavaScript. In ParallelJS, an application is written in the combination of regular JavaScript semantics and the constructs defined in a library. The program can be executed in either the CPU using the native JavaScript compiler or translated to PTX and executed in the GPU.

ParallelJS is evaluated with a set of JavaScript programs executed on systems with Intel CPUs and NVIDIA GPUs. This paper makes the following main contributions:

- A compilation and execution flow running JavaScript programs expressed in parallel constructs on heterogeneous systems with both a regular CPU and a massively-parallel GPU.
- A systematic performance evaluation and comparison of the implementation on different heterogeneous architectures with emphasis on understanding how to efficiently accelerate high-level programming languages utilizing massively-parallel processors.

The rest of the paper is organized as follows. Section 2 introduces the JavaScript language, GPGPU terminology and tool chains used in this paper. Section 3 provides a system overview. Section 4 describes the language extension defined in ParallelJS infrastructure. Section 5 proposes the implementation of ParallelJS on heterogeneous architectures which consists of a compiler frontend and a runtime system. Section 6 evaluates and compares the performance of different systems. Section 7 reviews the related work, followed by the conclusion in Section 8.

GPGPU-7 March 01 2014, Salt Lake City, UT, USA

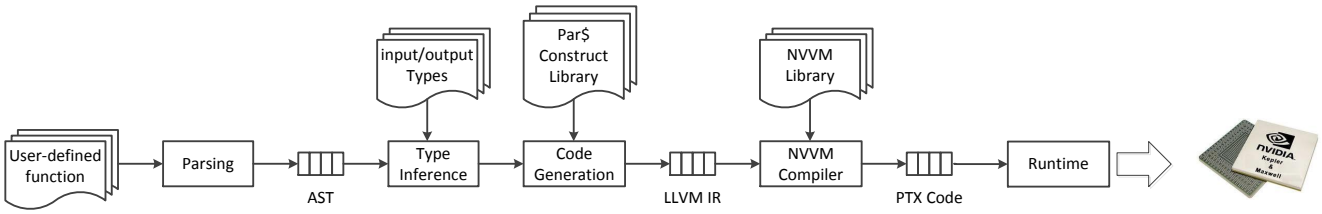

Figure 1: Compilation Flow of ParallelJS

2. BACKGROUND

2.1 JavaScript

JavaScript is a programming language that has been widely used in website clients, servers, mobile applications, etc [4]. It focuses on security and portability. Security means that JavaScript has been limited to execute in a sandbox so that calling C/C++ code from JavaScript requires additional extensions. Portability enables the same set of JavaScript programs to execute on different platforms without any system-dependent configurations or changes.

JavaScript is dynamically typed, does not have classes, and supports high level functions and closures. Programmers in JavaScript often use a rapid iterative development style which limits the use of offline compilation. JavaScript Engine is used to compile JavaScript to native binary running on CPUs. Optimizations on JavaScript engines in the mainstream browsers such as V8 in Google Chrome [5] and SpiderMonkey [16] in Mozilla Firefox enable highly efficient execution of JavaScript code on CPUs. Traditionally single threaded, JavaScript Engine is able to conserve the features of JavaScript such as determinism. However, single-threaded JavaScript execution fails to utilize the parallel architectures in modern processors.

2.2 NVIDIA GPU Compilation Tools

The implementation of ParallelJS in this paper targets NVIDIA GPUs and uses the compilation flow of NVIDIA's CUDA programming model. NVIDIA GPU compiler, NVCC, can compile a CUDA application [17] into PTX format, a virtual ISA that is realized on NVIDIA GPUs [19]. This PTX representation is a RISC virtual machine similar to LLVM [13], a widely used language and device independent Intermediate Representation (IR). The NVVM compiler [18] can translate LLVM IR into PTX, therefore many existing LLVM optimization and analysis passes can be directly used for GPUs.

While the current implementation is based on CUDA, the use of the Bulk Synchronous Parallel (BSP) model also permits relatively straightforward support for industry standard OpenCL [10].

3. SYSTEM OVERVIEW

A ParallelJS program uses the same syntax defined by regular JavaScript. The program is compiled, like a regular JavaScript program, by executing all the statements and functions invoked by a JavaScript procedure (e.g. a webpage or an application).

ParallelJS extends the language by defining a new library with a special root object `par$` which includes 1) a new basic data type `AccelArray` and 2) a set of utility constructs such as `par$.map` and `par$.reduce` that operates on `AccelArray`. These constructs can further call any regular JavaScript functions as user-defined functions. For example, construct `par$.map` can use addition operation as its user-defined function. Unlike low-level language bindings such as WebCL [9], ParallelJS is built directly into JavaScript so that programmers do not need to use a second language.

The ParallelJS compiler is responsible for converting the user-defined function along with the invoked `par$` utility construct to low-level binary code that executes on the target GPU. The compiler is designed to be used in Just-In-Time (JIT) compilation which is the same scenario for JavaScript engines to execute regular JavaScript code, i.e., when a `par$` construct is called from a JavaScript program, the runtime invokes the ParallelJS compiler to generate code for this construct.

Figure 1 shows the complete ParallelJS compilation flow, including four stages: Parsing, Type inference, Code generation and NVVM compilation. The first three stages parse and compile constructs to LLVM IR. Type inference is necessary to generate type information since JavaScript is untyped while LLVM is strictly-typed. Constructs are written in JavaScript themselves and delivered as part of the ParallelJS library. The final stage uses stock NVVM compiler to compile generated LLVM IR to PTX kernel. After JIT compilation, the runtime module manages the data allocation and movement as well as the kernel execution on GPUs. The detailed implementation for each stage are described in section 5.

ParallelJS is designed for heterogeneous systems that may or may not have an available massively-parallel processor such as a GPU. When the utility constructs are executed, the user-defined function can be applied in parallel using the GPU implementation of the constructs if there exists a GPU in the system. Otherwise the execution will fall back to a sequential implementation on the CPU. The runtime determines where the code runs.

4. LANGUAGE EXTENSION

The aim of ParallelJS is to enable JavaScript on GPUs with minimum change to the language itself. The language extension is to deal with two issues: i) hiding the low level memory hierarchy by introducing a new data type called `AccelArray`; ii) adding several constructs that allow developers to specify which functions may execute in parallel. Neither of these two extensions fundamentally change JavaScript semantics so JavaScript developers will be able to express data parallelism easily while still getting performance improvements.

Each `AccelArray` consists of an array of binary data along with helper information including sizes, types and

Construct	Alias	Description
AccelArray		Construct a new AccelArray from array indices.
map	collect, each, forEach	Map values of input AccelArrays through user-defined functions to output(s).
reduce	inject, foldl	Combines the values of an input AccelArray into a scalar result.
find	detect	Returns one of the values in the input AccelArray that pass a truth test.
filter	select	Returns all the values in the input AccelArray that pass a truth test.
reject		Returns all the values in the input AccelArray that fail a truth test.
every	all	Returns true if all of the values in the input AccelArray pass the truth test.
some	any	Returns true if any of the values in the input AccelArray pass the truth test.
scatter	shuffle	Reorders the input AccelArray by an integer index AccelArray.
gather		Gather values in the input AccelArray according to an integer index AccelArray.
sort		Sort the input AccelArray according to comparison function.

Table 1: Utility Constructs in ParallelJS

shapes. The helper information is specified explicitly when the AccelArray is constructed so that the compiler can determine the interpretation of the binary data correctly. Currently the supported data types include 8/16/32 bit integers and single/double precision floats. The binary data in AccelArray are organized as rectangular arrays whose shape is one, two, or three dimensional rectangular arrays. For example, a shape of [2,5] specifies that the output will contain 10 elements and should be treated as two-dimensional matrix of size 2x5.

ParallelJS adds a set of utility constructs with parallel semantics in par$ that define the operations on AccelArray. Each of these constructs can take one or more inputs and generate corresponding output AccelArray(s) or a scalar value. ParallelJS also requires the output data type specified explicitly for each construct (single precision float type by default). The construct can take any regular user-defined JavaScript function as one of their arguments. When executed, the construct applies the user-defined function to the input according to the operation specified by the construct.

Table 1 lists all the utility constructs supported by ParallelJS. Some constructs have aliases for the sake of convenience. Developers that want to add additional parallel operations can add new methods to the object par$.

In order to support parallel execution on GPUs, ParallelJS has the following requirements for the user-defined functions when used as an argument of the utility constructs:

Element-wise. The input arguments of the function correspond to one element of each input AccelArray of the construct. The constructs apply the element-wise function for each element in the input AccelArray(s). For some constructs such as par$.map, the function may also read elements of the whole input AccelArray using computed indices. While the function can read any location in the array, it is limited to write to only one place in its output array. For construct reduce, the function only defines the operation used for reduction (e.g., addition of two input elements) but does not generate any new element in the output array. When implemented on GPUs, the element-wise function will be mapped to operations of each GPU thread.

Side-effect Free. The function should be side-effect free, i.e., they cannot change any global state by writing into global variables or function arguments. This is important since ParallelJS should still preserve the original stability of JavaScript when accelerating it through GPUs, which means the result should be deterministic. The side effect-free functions guarantee that when called in parallel, they never cause race conditions.

No Order Restriction. The function should not have computation order restrictions so that they can be called in any order without changing the final result. This is important for parallel execution.

Syntax Requirement. The function can use arbitrary syntax when executed on CPU. However, when running on GPU, only a subset of the regular JavaScript syntax are supported, including JavaScript primitive data types number and boolean, basic control flow such as if/else, for/while loop. Complex JavaScript object, recursive control flow, closure functions are not supported. If the function uses syntax that is not recognized by GPU implementation, code is diverted to the CPU.

Scope Restriction. The function uses strict lexical scoping by not referring to any non-function identifiers that are defined outside the function body. However, the function can take a context object as its argument such that it contains all the identifiers from outside scope that are referred by the function body using this annotation. ParallelJS requires context objects can only have number, boolean, regular JavaScript typed arrays and AccelArray as the members.

Listing 1 gives an example of the par$.map construct which takes a simple JavaScript addition function. Line 3 defines a context object with one outside scope identifier length whose value is 100. Line 4 shows the element-wise function fn as the argument of construct map. It takes three arguments: a as the single input element, index as the index for this input element and array as the whole input AccelArray. Line 5 refers to the identifier length in context object by using this annotation prefix. Line 6 returns the output single element as the result of addition of the input single element and the next element of the input AccelArray, i.e. array[index+1]. Line 10 shows invocation of par$.map with arguments input, fn, context and generates the AccelArray output. Note that here the par$.map construct uses default data type par$.float32.

```
1  var arr = [...]; //A JavaScript array
2  var input = par$.AccelArray(arr, par$.float32);
3  var context = {length:100};
4  function fn(a, index, array) {
5    if(index < this.length - 1) {
6      return a + array[index + 1];
7    } else {
8      return a;}
9  }
10 output = par$.map(input, fn, context);
```

Listing 1: Example code of ParallelJS par$.map

While targeting for GPU, ParallelJS provides a higher programming level abstraction than the traditional GPGPU

programming languages such as OpenCL and CUDA. It does not expose any Bulk-Synchronize Parallel concepts such as Thread, CTA or Grid to the programmers. The memory hierarchy (global, shared and local memory) is hidden in the implementation details and transparent to the users. Furthermore, the algorithm scheme for each utility construct are low-level details that the programmers are not expected to deal with. Therefore, ParallelJS programs are typically far smaller then CUDA codes. Together with the flexible execution model on both CPUs and GPUs, ParalellJS provide performance portability while programmers work on the high-level parallel constructs.

5. IMPLEMENTATION

This section discusses the major components of implementing the ParallelJS framework. ParallelJS has implementation for the `par$` library both on the CPU and the GPU to enable portability. While mainly targeted for heterogeneous systems, programs written in ParallelJS can be executed on systems with or without CUDA-enabled GPUs. Furthermore, if ParallelJS programs fail running on GPUs due to unsupported syntax (e.g. not satisfying the requirement for the user-defined function as described in section 4) or runtime errors, they can be diverted to the CPU and report any errors from there.

Just as in regular JavaScript, the CPU implementations of ParallelJS utility constructs are sequential. Every construct is implemented as a loop in which the user-defined function applies to each element in each iteration. For example, `par$.map` construct takes one element from the input `AccelArray` indexed by the iteration number, feeds it into the element-wise function, generates one element and writes it to the output `AccelArray` at the same index.

On the other hand, in the GPU implementations, the constructs passes through a series of compilation stages that progressively lowers them into PTX kernels that are executed in parallel by an implementation of the runtime on the GPU. The rest of the section describes the compiler and runtime systems involved in this procedure.

5.1 Compiler

```
1  arguments[0]: a
2  arguments[1]: index
3  arguments[2]: array
4  body[0]:
5    BlockStatement:
6      IfStatement:
7        test: BinaryExpression:
8          left: index
9          right: BinaryExpression:
10           left: MemberExpression:
11             object: ThisExpression
12             property: length
13           right: 1
14       consequent: ReturnStatement:
15         argument: BinaryEpxression:
16             left: a
17             right: MemberExpression:
18               object: array
19               property: BinaryExpression:
20                 left: index
21                 right: 1
22       alternate: ReturnStatement:
23         argument: a
```

Listing 2: ParallelJS AST example

Parsing.

The parser of ParallelJS is based on the open source project Esprima [7]. It parses the user-defined function and generates an Abstract Syntax Tree (AST). The AST is stored as a JavaScript object whose children members are also JavaScript objects corresponding to statements or nodes.

Listing 2 shows the AST for the user-defined function `fn` in Listing 1 line 4-9. The AST begins with three function `arguments`, followed by a single function `body`. Each line from 5-23 is an object representing either a statement (e.g. `IfStatement` in line 6) or a node in a statement (e.g. `test`, `consequence` and `alternate` nodes for the `IfStatement` in line 7, line 14 and line 22 respectively). The node itself can be another statement (e.g. the `test` node in line 7 as `BinaryExpression`). The identifiers or literals referred to in the function body are leaves of the AST (e.g. **`index`** in line 8).

Type Inference.

The type inference stage uses the type information of the function input arguments and the identifier in the context objects as the starting point and propagates the type information to each statement to infer types for all the local variables and expressions. Since this stage is performed online, the types of all function inputs are known, which allows the system to avoid the iterative Hindley-Milner [8] [15] algorithm. Instead, the system iterates over the set of statements in the function that assign to local variables and then with all local variables typed, it does a single iteration to find types for all expressions. Recall that the input arguments of the constructs are always `AccelArrays` with their type information explicitly stored. Therefore, the type information of the function input arguments can be directly obtained from the input arguments of the construct. Similarly, the type information of the output argument is also available through the data type argument of the constructs. Numbers in the context object as well as the literals in the function body will be given a type according to the values.

The type inference procedure can be illustrated in an example for the `BinaryExpression` in Listing 2 line 9. The identifier `length` in line 12 is an integer for its value is 100. The property node in line 12 propagates to the `MemberExpression` in line 10 so that the `left` node of the `BinaryExpression` in line 9 gets the integer type. The right node in line 13 is a literal `1` which is an integer. Therefore, the type inference concludes the `BinaryExpression` has integer type.

It should be noted that the nodes of an expression can have different types (e.g., `BinaryExpression` has an integer `left` node but a float `right` node). The type inference will use the data type with maximum precision in this case for the expression type (e.g., float should be the type of `BinaryExpression` for the above example).

Code Generation.

Code generation takes the typed AST as well as the utility construct name to generate LLVM IR, the format of which is compatible with the NVVM compiler requirement. The procedure consists of four steps:

Step 1: Add kernel header and meta data. The kernel header and meta data include some key words to define a GPU kernel in the LLVM IR so that they can be recognized by the NVVM compiler. Listing 3 line 1 and line 17-19 shows

an example of kernel header and meta data for the kernel `map`.

Step 2: Generate kernel arguments. The kernel arguments include all the `AccelArray(s)` in the input, output, context object and the `numbers`, `booleans`, regular arrays in the context object. Arrays and `AccelArrays` are represented by data pointers while `numbers` and `booleans` are scalar identifiers. Listing 3 line 2-4 shows the three kernel arguments for the `map` example in Listing 1: `output` and `input` corresponding to the output and input `AccelArrays`, `length` corresponding to the identifier in context object.

```
1  define void @ map(
2      float *output,
3      float *input,
4      i32 length) {
5      %tid = ...compute tid...
6      %aptr = getelementptr float* %input, %tid
7      %a = load float* %aptr
8
9      %r1 = add i32 %tid, 1
10     %r2 = getelementptr float* %input, %r1
11     %r3 = load float% %r2
12     %out = fadd %a, %r3
13
14     %outptr = getelementpr float* %output, %tid
15     store %out, float* %outptr
16 }
17 nvvm.annotations = !{!1}
18 !1 = metadata !{void(...)*@map, metadata
19   !"kernel", i32 1}
```

Listing 3: Example of LLVM IR generated for `map` by ParallelJS code generator

Step 3: Generate LLVM instructions for data preparation and finalization. In this step, the code generator generates the instructions for reading the input data and writing the output data according to the utility construct invoked. For example, for the `map` construct, the input data should be one element from the input array indexed by the thread id and the output element should be stored in the output array indexed by the thread id. Listing 3 line 5-7 and line 14-15 show the input loading and output storing. Ellipses (...) in line 5 indicate the computation of global thread id using thread/block/grid ids depending on the shape of input `AccelArray`. ParallelJS requires the shape of all the input `AccelArrays` to be identical, so the thread id generated here can be universal for all the inputs.

```
1  define void @ reduce(...) {
2      ... ;compute tid
3
4      ;load data from shared memory
5      %s= load float addrspace(3)* %saddr
6
7      ... ;computation for reduction
8
9      ;store result to shared memory
10     store %out, float addrspace(3)* %saddr
11 }
```

Listing 4: Example of LLVM IR generated for `reduce` by ParallelJS code generator, where `addrspace(3)` represents shared memory space in LLVM IR

The algorithm skeletons used for each construct are based on the state-of-art implementation. ParallelJS maintains a set of LLVM IR code for each construct. The implementations can be architecture-dependent to achieve performance portability, which means adaptive decision of kernel configuration such as Grid/CTA size, local/shared/global memory usage. For example, construct `reduce` uses shared memory to store the temporary computation result in its CTA-wise $\log N$ stages as shown in Listing 4. This kernel will also be generated multiple times to handle the inter-CTA $\log N$ parallel stages.

Step 4: Generate LLVM instructions from each statement. This step walks through the typed AST, examines each statement or node and generates LLVM IR instruction(s) for them. Data type casting instructions are inserted if necessary. External functions will be mapped to LLVM IR intrinsics. ParallelJS supports limited external functions such as mathematics/logic operations (e.g. trigonometric functions).

Line 9-12 of Listing 3 shows the LLVM IR instruction generated for line 6 in Listing 1. The `return` statement is replaced by storing the result into `%out`.

NVVM Compiler.

The last stage of the ParallelJS compiler uses the NVVM compiler to compile LLVM IR to PTX kernel(s). The NVVM compiler is shipped as a library which defines a set of C APIs that are directly invoked by ParallelJS framework in its runtime systems.

5.2 Runtime

The runtime executes PTX kernels on the GPU and manages the input/output data exchange between the CPU and GPU. The runtime system is written in C/C++ along with CUDA driver APIs and shipped as a dynamic library. ParallelJS invokes the runtime library directly from the JavaScript program, which requires the setup of a secure interface between them. We use the privileged JavaScript interface provided by the Firefox browser extension. For other browser or applications, the idea should be the same although the interface implementation could be different.

Figure 2 illustrates the interface between JavaScript and the runtime library. LLVM IR, input and context object that resides on the regular JavaScript side pass to runtime through Privileged JavaScript. Firefox uses Document Object Model (DOM) event object to send and receive the passed data. The DOM event is created in the regular JavaScript and dispatched with its event name. On the privileged JavaScript side, there is an event listener that can take this DOM event and invoke corresponding procedures in the runtime library. Similarly, when the runtime procedures terminate, the privileged JavaScript can send back event to regular JavaScript indicating output data are ready to retrieve. It should be noted that the NVVM compiler is also invoked by the runtime library through the privileged JavaScript interface.

Data Management.

The implementation of the runtime system uses a flat array to represent the data stored in `par$.AccelArray`. ParallelJS transfers the pointer of the `AccelArray` data array to the runtime which is then copied to GPU memory. For `number` and `boolean` values, the corresponding values are set to kernel arguments directly. After kernel execution, runtime copies the output data from GPU back to the `par$.AccelArray`.

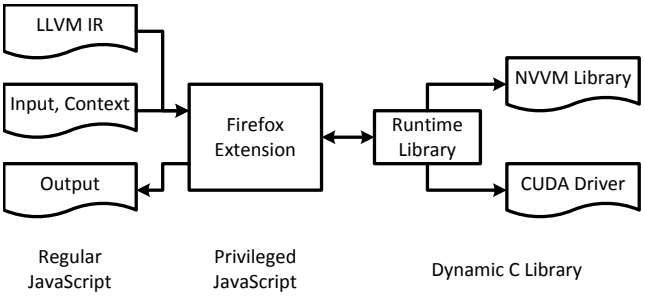

Figure 2: Runtime Interface Overview

ParallelJS uses a data management system that minimizes the data transfer between the GPU and CPU. The system copies the data in an `AccelArray` to the other side only if there is a requirement to access the data in that side. For example, if a ParallelJS program calls two back-to-back `map` constructs but only reads the output of the second `map`, the result of the first `map` can reside in the GPU without transferring back to the CPU. The data management system is also designed to collaborate with the JavaScript garbage collector to reuse the system memory efficiently. Once the garbage collector releases an `AccelArray`, the corresponding data stored on the GPU will be released accordingly or reused by a new `AccelArray`.

Kernel Execution and Code Cache.

PTX kernels generated by the ParallelJS compiler execute on GPUs by calling CUDA driver APIs, including arguments setup, kernel configuration and kernel launching. ParallelJS maintains all the kernel handlers for each PTX kernel generated. These kernel handlers are used in a code cache to enable faster compilation and kernel reuse. When a construct with a user-defined function is executed, its name, the user-defined function code and input/output/context variable types are stored as a hash code. The hash code is mapped to a PTX kernel handler in the runtime. Any future invocation of the same construct with the same hash code does not require any compilation but can use the stored kernel handler instead. This will significantly reduce the compilation time especially for programs in which the same set of operations are called within a loop.

6. EXPERIMENTS AND EVALUATION

We execute the ParallelJS infrastructure on a heterogeneous system with both a CPU and GPU. We choose a high-end desktop and a low-end business class laptop as the experimental environment specified in Table 2. The experiments are performed on Windows OS as it has a better support for the Mozilla Firefox privileged JavaScript interface. The CPU implementation of the ParallelJS utility constructs are evaluated on both Chrome and Firefox browsers with the stock JavaScript engines while the GPU implementation is only evaluated on Firefox for its support of privileged JavaScript to invoke an external C library. We also compare the code generated from ParallelJS with native CUDA code written for the example program as well as those shipped as libraries (e.g. CUB library from NVIDIA [14]).

6.1 Example Programs

We investigate the performance of ParallelJS code in several example programs. The core computation part of the original JavaScript code is rewritten using the ParallelJS utility constructs.

	Laptop	Desktop
GPU		
GPU (NVIDIA)	GTX 525M	Geforce Titan
Architecture	Fermi	Kepler GK110
Cores	96	2688
Device Memory	1024 MB	6144 MB
Clock Freq.	600MHz	837MHz
CPU		
CPU	Intel i7-2630QM	Intel i7-4771
Clock Freq.	2.0GHz	3.5GHz
System Memory	6GB	32GB
OS	Windows 8	
Browser	Mozilla Firefox 22.0	
	Google Chrome 31.0	
CUDA and NVVM	5.5	

Table 2: Experimental environment.

Boids. The boids example simulates the flocking behavior of birds in lock step. The birds are moving in the space such that they avoid any collision with other flockmates while steering towards the average position of local flockmates. The original JavaScript implementation uses a loop in which the position of each flockmate is computed according to the rules. A switch variable can be turned on or off by a mouse to control whether the flockmates either attract or repel each other.

Chain. The chain example simulates a set of springs which are connected to each other reacting to any mouse dragging. When parts of the spring network are dragged to a random point in the screen, the remaining part will react and adjust their positions according to the forces placed on the springs. The computation of the spring force includes all the forces from neighbor nodes adjusted by spring configurations such as stiffness.

Mandelbrot (Mandel). This example computes the Mandelbrot set recursively such that in each iteration the boundary is elaborated to progressively finer detail. The original JavaScript code uses a loop in which every pixel of the image is computed according to the Mandelbrot equation.

Reduce. This is a simple reduction program using addition as the reduce operation. It uses the `par$.reduce` construct.

Single Source Shortest Path (SSSP). The ParallelJS implementation of SSSP has two steps: 1) it does a backward walk on the graph, loop over all the predecessors of each node to find the minimum distance. 2) it checks the number of the nodes whose values change. The procedure terminates when there is no change in the graph. This example uses both the `par$.map` and `par$.reduce` construct.

The examples **Boids, Chain, Mandel** have both a computation part and a visualization part, which is a common pattern for lots of JavaScript programs on webpages. They share the similar code structure, which uses an outside loop for time-step simulation. In each iteration, each node or pixel compute its new position or value according to some equation which may or may not involve its neighbor data. The pattern makes these example good candidate for ParallelJS `map` construct. The user-defined function of `map` would be a computation for each node/pixel.

```
1  //sequential JavaScript code
2  function seqfn(...) {
3    for(var x = 0; x < width; x++) {
4      for(var y = 0; y < height; y++) {
5        var xy = ...computation of point (x,y)...
6        result[y*width+x] = xy; }
7    }
8  }
9
10 //ParallelJS code
11 function parfn(input, index) {
12   var xy= ...computation of point (x,y)...
13   return xy;
14 }
15 par$.map(input, seqfn, context);
```

Listing 5: Example JavaScript program written in ParallelJS using `par$.map`

Listing 5 shows how the patterns in these three example programs are mapped to ParallelJS code. Generally the computation inside the loops of the sequential code (line 5-6) will be mapped directly to the body of the user-defined function (line 12-13). This strategy provides productivity of ParallelJS so that the JavaScript programmers do not have to rewrite the program from scratch to get the benefit of parallel execution.

```
1  // create a parallel array of size sz
2  var p = new par$.AccelArray(sz,fill);
3
4  // log n binary reduction done in shared memory
5  var f = par$.reduce(p, sum);
```

Listing 6: Example ParallelJS program **Reduce**

Listing 6 shows the ParallelJS code for **Reduce** with the sum operation. Note that internally `par$.reduce` is implemented with the log N parallel scheme using shared memory as computation storage as shown in Listing 4.

Part of the ParallelJS implementation of **SSSP** is shown in Listing 7. The `par$.map` takes `mapfn` as an argument which checks the cost of neighbors of every node and find minimum distance. The `par$.reduce` gathers changed flag for every node using addition function. This piece of code is compared directly with **SSSP** code written in CUDA [12] and uses a significant smaller number of lines (40 lines in JavaScript versus 150 lines in CUDA).

```
1  while (s){
2    par$.map(inData, outData, mapfn, ctx);
3    ...
4    s = par$.reduce(pChanged, sumfn);
5  }
```

Listing 7: Part of **SSSP** code written in ParallelJS

6.2 Evaluation of the GPU Implementation

We evaluate the performance of the GPU implementation of ParallelJS for the first four example programs on the desktop/laptop GPUs by comparing the speedup over the sequential implementation on Intel CPUs. We measure the execution time of one time-step simulation averaged from 10000 iterations and compute the speedup over the sequential CPU implementation on Intel Haswell i7-4771. Data transfer time is included in the total execution time since the CPU requires to access the output `AccelArray` after each iteration. Code cache is enabled so the compilation

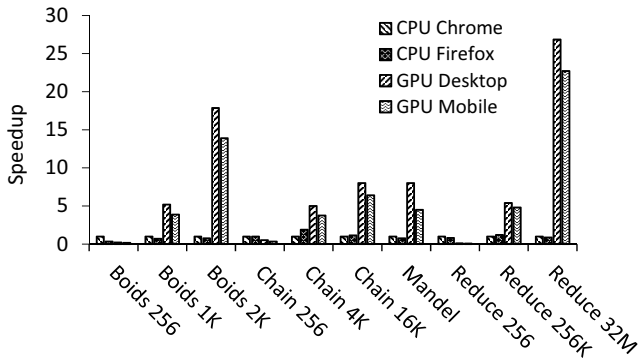

Figure 3: Performance of ParallelJS GPU implementation which shows the speedup of GPU implementation on Desktop/Laptop GPUs over the Desktop CPU sequential implementation. The CPU implementation on Chrome browser sets the baseline. Number suffix in the x-axis indicates the input size for each example.

only happens at the first iteration and its timing consumption is excluded. Rendering time is excluded from the measurement. The CPU implementation are measured both on Chrome and Firefox browsers. The results are shown in Figure 3 where CPU Chrome is the baseline. The time measurement here is performed by calling JavaScript timing function.

The CPU implementation on the Chrome browser and Firefox browser achieve similar performance across the four examples with different input sizes. For small input size (Boids 256, Chain 256 and Reduce 256), the GPU implementation on both the laptop and the desktop platforms performs worse than the sequential implementation. The reason is that for small input sizes, the overhead of compiling ParallelJS, communication between CPU and GPU, as well as kernel launching takes most of the total time. Speedup can be observed on GPUs for larger input (Boids 1K and 2K, Chain 4K and 16K, Mandelbrot, Reduce 256K and 32M). The desktop GPU achieves 5.0x to 26.8x speedup while the laptop GPU achieves 3.8x to 22.7x over the baseline.

The speedup over the sequential implementation demonstrates the efficiency of the ParallelJS infrastructure utilizing the massively-parallel data processor in the heterogeneous systems. The difference between the speedup on the laptop GPU and the desktop GPU shows that the latter has much stronger computation power due to larger number of cores, advance in architecture and higher clock speed.

To further analyze the performance result, we break down the compilation time and execution time for one iteration of *Boids 2K* on laptop GPU as shown in Table 3. ParallelJS compilation time includes parsing, type inference and code generation as described in section 5.1. NVVM compilation time is spent on invoking the NVVM library to generate PTX kernels. If code cache is enabled, the time spent on these two parts only happen at the first iteration where a new kernel should be generated for the construct. However, there are still cases that the same construct cannot be reused and the compilation time consumption can cause poor overall performance. Therefore, optimization in compilation may still be necessary. The privileged JavaScript interface cost 1ms which is spent on passing DOM event object between regular JavaScript and privileged JavaScript.

Stage	Time (ms)
ParallelJS Compilation	13†
NVVM Compilation	14†
Privileged JavaScript interface	1†
Data Transfer between CPU and GPU	1.92‡
Kernel Execution	5.82‡

Table 3: Execution and compilation break down of Boids2K GPU implementation. † indicates time is measured using JavaScript timing API and ‡ indicates time is measured using C API.

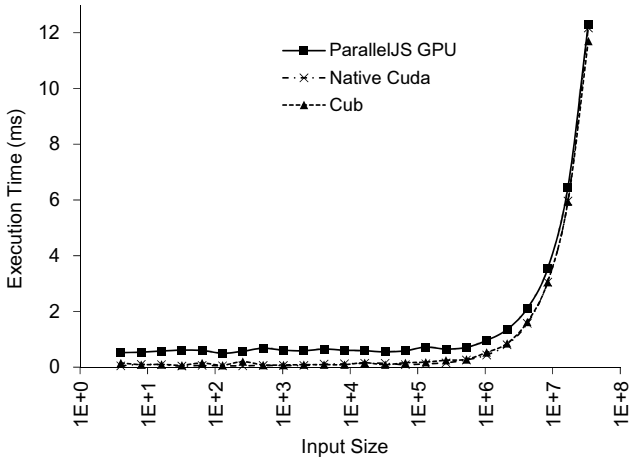

Figure 4: Comparison of the performance of **Reduce** program in ParallelJS, native CUDA and CUB library

The time consumption is independent of the program size. The data transfer time takes about 22% of the total execution time. However, this part can be avoided if the CPU does not require output data access in every time step. In the first four examples, CPU needs the output data to write the display frame buffer. In the future, ParallelJS can be designed such that GPU can directly display data from the `AccelArray` which resides on the GPU. In that case, no data transfer between CPU and GPU is necessary. The kernel execution composes most of the total execution time (66.6%), which generally depends on the algorithm used by the user-defined function and the construct. ParallelJS is designed such that the implementation of the constructs can be flexible depending on the GPU architecture it targets for. Therefore, the kernel execution time can be optimal for different systems.

Figure 4 shows the performance of Reduce example in ParallelJS versus native CUDA code and CUB library. Similarly, the overhead in ParallelJS framework takes up most of the execution time for small-size inputs, making the performance a lot worse than native CUDA or CUB (10x and 7.6x slow down for 1K input size). For larger input size, ParallelJS performs similar to but still a little worse than native CUDA or CUB (1.01x and 1.05x slow down for 32M input size) due to the extra layer introduced by the framework on top of the CUDA binary. However, the high level design methodology of ParallelJS enables incredible smaller code size compared with native CUDA or CUB. This raises the question of how the current software stack balance the tradeoff between productivity versus performance. While ParallelJS provides high programmability by hiding lots of GPU details such as Grid/CTA sizes and memory hierarchy, certain low-level customary can result in better performance. We are still investigating cases where the proposed framework can perform well or not compared with native GPU implementations.

Graph	nodes	edges	Time (ms)	Speedup
USA-Road	1070376	2712798	18685	0.86
r4-2e20	1048576	4194304	1215	0.60
rmat20	1048576	8259994	3645	0.59

Table 4: Performance of SSSP on ParallelJS

We evaluate the performance of SSSP by comparing its ParallelJS implementation with CUDA implementation [12]. We report the speedup over CUDA implementation on several graphs with different number of nodes and edges. The result is shown in Table 4. The performance of SSSP is always worse than the CUDA version (speedup from 0.59x to 0.86x). The main reason is that ParallelJS uses the backward algorithm instead of the forward algorithm in the original CUDA implementation. This is inevitable since in the original CUDA implementation, atomic operations are used to find the minimum distance while JavaScript is a deterministic language without any support of atomic operations and has to use a backward walk on each node to loop over all the predecessors. This example shows the limit of current ParallelJS framework compared with the stock programming language. Some low-level features are not visible to the programmers in ParallelJS, thereby eliminating the ability of ParallelJS to support very complex GPU applications efficiently. A second issue is that the lonestar code does all the work in a single kernel while JavaScript code needs two kernels – one to update the nodes and a second to detect any node changes. In the future, we may be able to fuse these kernels together to improve the performance.

6.3 Discussion

The performance of ParallelJS shows that the implementation on both desktop and laptop GPUs are efficient. The same infrastructure can be easily moved to future mobile platforms which are becoming more prevalent for web browsing, hence running JavaScript programs. Since mobile platform might have lower-end GPUs compared to desktops and laptops, the performance evaluation of ParallelJS on such platforms are useful and necessary. Based on that, people can explore the research topics such as choosing right constructs to build the web applications for different device platform.

The fused CPU/GPU [1] has been a recent trend for heterogeneous systems. The shared memory hierarchy in these fused architectures would generate impact on ParallelJS design methodology especially for the data transfer part. While a memory management would still be necessary, share memory between CPU and GPU would certainly reduce the memory traffic and thereby improve the overall performance. In this case, an intelligent CPU-GPU co-scheduling method is necessary to decide where the construct should run.

7. RELATED WORK

Traditional JavaScript is single threaded using asynchronous events that multi-threaded shared memory models

are not robust enough for web programming. While there is better performance, data races, locks, timing issues, live locks etc., may lead to cross browser failures and a poor user experience. A feature called Web Workers [21] is the only widely adopted parallel model avoids all of this issues by offloading all long running computations to background threads with an extremely heavy weight cross thread communication mechanism. The overhead of cross thread communication makes them unsuitable for scalable high performance parallel computing.

People have been making effort to improve the JavaScript performance by utilizing GPUs. The WebCL [9] proposed by Khronos Group and the JSonGPU framework [20] proposed by Pitambare et al. are among the earlier researches that target JavaScript on GPUs. The former defines a JavaScript binding to the OpenCL standard for heterogeneous parallel computing. The latter proposes extension to the JavaScript language such that the programmers can express the BSP model directly with JavaScript code. While all the above works are useful and report significant performance over sequential JavaScript execution, they both expose the details of the GPU programming concepts to JavaScript by introducing new syntax and semantics. Therefore, programming with these frameworks requires understanding of the GPU programming models. Furthermore, one has to investigate the code for different GPU architectures to achieve best performance. The proposed ParallelJS framework hides away all the GPU programming details from the programmers and stays at higher level constructs while extending the JavaScript language, which enables less programming effort as well as performance portability since the implementation of constructs can be adapted automatically according to different architectures the programs execute on. Intel proposes the River Trail [6] framework to execute JavaScript on multi-threaded CPUs by introducing parallel primitives with an OpenCL backend. However, RiverTrail does not run on GPUs and lacks the design perspectives of handling GPU-specific features such as data transferring between CPU and GPU.

As a recent trend of heterogeneous computing, executing high-level programming languages or domain-specific languages (DSL) on heterogeneous architectures with GPUs are drawing great attentions in different research communities. For example, Wu et al. [22] designed Red Fox compilation framework that can run relational queries on GPUs. Klöckner et al. develop the PyCuda framework [11] and Catanzaro et al. develop the Copperhead framework [3], both for executing Python script language on GPUs. The former uses strategy that is very similar to WebCL, which provides an interface for directly invoking external CUDA code from Python. The latter compiles a Python program with parallel primitives to Thrust code [2] and links it against the Thrust library to run on GPU. Compared with these frameworks, ParallelJS provides higher programmability by both eliminating CUDA code embedding and automating the decision for where to run the JavaScript programs (either on CPUs or GPUs).

8. CONCLUSION

This paper presents the design of a GPU compiler/runtime framework, ParallelJS, for JavaScript which is commonly used for web applications. The paper focuses on the compilation of commonly used constructs onto GPUs. Comparison with mainstream host implementations demonstrates significant computing speedup is feasible. The language is progressively parsed and lowered through a series of IR representations that eventually are mapped to PTX kernels that are executed by an implementation of the runtime on a discrete GPU. When compared with a CPU-based mainstream JavaScript engine, ParallelJS system can be up to **26.8x** faster. We also provide analysis of the performance, the lessons learned and future directions.

9. REFERENCES

[1] AMD. Amd fusion family of apus: Enabling a superior, immersive pc experience. March 2010.

[2] N. Bell and J. Hoberock. Thrust: A 2 6. *GPU Computing Gems Jade Edition*, page 359, 2011.

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. *SIGPLAN Not.*, 46(8):47–56, Feb. 2011.

[4] D. Flanagan. *JavaScript*. O'Reilly, 1998.

[5] Google. V8 javascript engine: Introduction. 2010.

[6] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: A path to parallelism in javascript. *SIGPLAN Not.*, 48(10):729–744, Oct. 2013.

[7] A. Hidayat. Esprima: Ecmascript parsing infrastructure for multipurpose analysis. http://esprima.org/.

[8] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[9] Khronos. Webcl working draft. 2013.

[10] Khronos Group. The OpenCL Specification, version 2.0. November 2013.

[11] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, A. Sarma, D. Nanongkai, G. Pandurangan, P. Tetali, et al. Pycuda: Gpu run-time code generation for high-performance computing. *Arxiv preprint arXiv*, 911, 2009.

[12] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[13] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04, pages 75–. IEEE Computer Society, 2004.

[14] D. Merrill. Nvidia cub library. 2013.

[15] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[16] Mozilla. Spidermonkey. 2013.

[17] NVIDIA. Cuda c programming guide, 2012.

[18] NVIDIA. Nvvm ir specification 1.0. 2013.

[19] NVIDIA. Parallel thread execution isa, 2013.

[20] U. Pitambare, A. Chauhan, and S. Malviya. Just-in-time acceleration of javascript.

[21] W3C. Web workers, 2013.

[22] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. CGO '14, 2014.