River Trail: A Path to Parallelism in JavaScript

Stephan Herhut Richard L. Hudson Tatiana Shpeisman Jaswanth Sreeram

Intel Labs
Santa Clara, CA, USA
stephan.a.herhut, rick.hudson, tatiana.shpeisman, jaswanth.sreeram@intel.com

Abstract

JavaScript is the most popular language on the web and is a crucial component of HTML5 applications and services that run on consumer platforms ranging from desktops to phones. However, despite ample amount of hardware parallelism available to web applications on such platforms, JavaScript web applications remain predominantly sequential. Common parallel programming solutions accepted by other programming languages failed to transfer themselves to JavaScript due to differences in programming models, the additional requirements of the web and different developer expectations.

In this paper we present River Trail — a parallel programming model and API for JavaScript that provides *safe*, *portable*, *programmer-friendly*, *deterministic* parallelism to JavaScript applications. River Trail allows web applications to effectively utilize multiple cores, vector instructions, and GPUs on client platforms while allowing the web developer to remain within the environment of JavaScript. We describe the implementation of the River Trail compiler and runtime and present experimental results that show the impact of River Trail on performance and scalability for a variety of realistic HTML5 applications. Our experiments show that River Trail has a dramatic positive impact on overall performance and responsiveness of computationally intense JavaScript based applications achieving up to 33.6 times speedup for kernels and up to 11.8 times speedup for realistic web applications compared to sequential JavaScript. Moreover, River Trail enables new interactive web usages that are simply *not even possible* with standard sequential JavaScript.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming — Parallel Programming; D.3.2 [*Programming Languages*]: Language Classifications — JavaScript; D.3.4 [*Programming Languages*]: Processors — Compilers

Keywords Parallelism; JavaScript

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509516

1. Introduction

The power wall has put a hold to the ever increasing growth of processor frequencies, forcing hardware manufactures to look for alternative ways to improve program performance. Modern microprocessors for all form factors routinely feature multiple cores, vector instructions, and tightly coupled graphics processing units. Parallel hardware has become a commodity. In response, language extensions or APIs for parallelism have emerged for most popular programming languages, slowly pushing parallel programming into the mainstream; however, one popular programming language, JavaScript, the *lingua franca* of the web, has surprisingly escaped this trend and remains predominantly sequential.

JavaScript has long outgrown its original use as a lightweight scripting language for web pages [2]. Increasingly, it is used for large scale applications that have as big a potential to benefit from parallel execution as any other client-side application. In combination with HTML5 [4], JavaScript is the sole universally supported programming environment for browser-based web applications; it is also emerging as a full-scale software development stack for stand-alone applications, especially in the mobile space [20]. HTML and JavaScript applications are rapidly gaining access to the capabilities traditionally reserved for native applications, such as fast 2D and 3D graphics, offline storage, video and audio content, camera capture and geo location, to name just a few.

With these new capabilities come new usages and application scenarios, of which many quickly outgrow the performance envelope offered by the current, sequential JavaScript implementations. In the context of 3D graphics, operations like skinning for character animation, collision detection for gaming or simulation of particle systems come to mind. Access to the camera built into a user's device enables new applications from in-browser video conferencing to novel ways of human device interaction. In both scenarios, processing the video stream in (near) real time is essential. Fast 2D rendering enables new forms of data visualization,

which, in particular in combination with large data sets, require speedy layout computations. For these and many yet to be conceived applications, access to parallel hardware and its promised performance gain will be required to realize the web platform's full potential.

The open web has a unique combination of requirements that preclude the use of existing parallel programming APIs. First is the hard requirement of safety and security. In contrast with most platforms, the user has little control over what code is executed in the browser. Applications are run by simply visiting a web page. Introducing additional malware attack surfaces is simply not an option. Second is the web developer community's demand for a single familiar programming model that builds upon the knowledge and tools the developers already possess. In particular, programmers have grown used to deterministic program behavior, which is typically undermined by concurrent programming. Third is the need to support a wide range of form factors and hardware architectures. A parallel programming model has to be generic enough to support multi-core CPUs, vector instruction sets, and programmable GPUs from a single source base. This requires a higher level of abstraction than commonly found in existing APIs. Finally, for the model to be successful, it should be capable of extracting dramatic performance improvements from the available parallel hardware.

In this paper we present River Trail, our vehicle to explore the design space for a parallel programming API that addresses those four challenges. We have used the same language and API design model that HTML5 and JavaScript are successfully applying to move the web forward: rapid prototyping of APIs and a lively language designer/application developer feedback cycle. To enable the former, we have implemented River Trail as a sequential library on top of JavaScript that runs in all modern browsers. This allowed us to quickly evolve the API with minimal development overhead while covering most client systems. We also implemented a prototype compiler on top of OpenCL to be able to evaluate our design against the fourth requirement: performance. At the same time, we have gathered as much developer feedback as possible. We have developed our prototype in the open on GitHub[1] to give developers early access. We also recruited a group of developers ranging from college students to professional web developers to gather early feedback during the design of the API. This helped us particularly in addressing the second challenge: providing a comfortable API.

Another advantage of involving developers early on is that it helped seed our set of benchmarks. Bringing parallelism to the web is not an incremental change. Instead, we aim to change the way users experience web applications. Existing benchmark suites such as SunSpider, V8 and Kraken do not reflect real web applications [26] and web applications are typically well tuned to stay within available performance characteristics. Instead, our benchmarks, including for instance real time video processing, motion tracking and game physics, look at applications that are not possible in browsers today. The more recent Octane suite consists of programs taken from V8 and several other programs that have been compiled from C/C++ to JavaScript. We believe that these programs do not represent how JavaScript is actually used by programmers for developing web applications either - the applications we consider are all developed in JavaScript.

Our design has converged on an API[2] centered around a single new data type *ParallelArray* as a basic abstraction for parallel computation, accompanied by six methods that implement well known parallel patterns: map, reduce, scan, scatter and filter. The ParallelArray API provides the high-level of abstraction necessary to support multiple platforms and hardware parallelism flavors, such as multiple cores, vector instructions and GPU executions. Our API provides deterministic parallelism for the majority of programs. We guarantee deterministic execution equivalent to a sequential execution except for scan and reduce operations that use non-commutative or non-associative operations. This level of determinism is similar to that of Google's map-reduce programming model [9] and seems to be an acceptable trade off well received by programmers. Finally, our API trivially provides the same level of safety and security as JavaScript because it stays within the boundaries of the same programming language.

We have evaluated our API with multiple web applications that demonstrate realistic usage scenarios such as web-based gaming, real-time video effect filtering and 3D computer animations, and several kernels. Most applications were developed by external parties. Our experiments show that River Trail is capable of delivering an order of magnitude performance improvement on an off-the-shelf system for a realistic web application, and even larger, up to 33.6x times, performance improvements for fully parallelized kernels. These speedups are sufficiently large that they not only improve the quality of user experience (e.g., less jitter while applying a video effect) but also enable applications that are simply performance infeasible without River Trail, such as Bugs — an interactive gesture based browser game. River Trail, thus, delivers on our objectives: a simple programming model with dramatic performance improvements.

The rest of this paper is organized as follows. Section 2 describes River Trail language design and APIs. Section 3 presents the implementation of the River Trail prototype. Section 4 presents the experimental results. We describe the related work in Section 5 and conclude in Section 6.

[1] `http://github.com/RiverTrail/RiverTrail`

[2] We have further distilled our results into Parallel JavaScript [15], a proposal for a parallel programming API put in front of the ECMA TC39 committee, which is tasked to evolve the JavaScript standard.

2. Language Design

River Trail was designed from ground up to meet the requirements of the open web. To fulfill the web's safety requirements, we have designed River Trail to enable the use of the same safety mechanisms that typical JavaScript implementations use. This includes, for instance, the absence of pointers, bounds checked array accesses, and automatic heap management.

River Trail also makes use of high-level parallel patterns which we believe are a requirement to enable platform portability. Like JavaScript, we refrain from exposing device specific types and use a generic number type instead.

Last but not least, River Trail puts ease of use over performance. Designing for web developers' productivity for us meant two things: Reducing the likelihood of programmer errors and keeping the annotation burden due to parallelism low, both while achieving performance at an acceptable level. The next two sections elaborate on our approach.

2.1 Concurrency Model

Various programming models aim to harness the concurrency available in modern hardware. Shared memory models commonly found in languages like Java, C# or C++ use mutation of global state as means for communication between concurrent threads. Such models come with a host of programming hazards such as complicated memory models, data races, dead lock and live lock. These concepts were deemed sufficiently dangerous to the web development community that they were rejected outright as a non-starter for language design.

On the other hand, Actors and other message passing based programming models and languages segregate both computation and data and use messages to communicate. Some of the better known examples include Actors and Erlang. HTML5 also adopted this approach in the form of web workers. They were designed for offloading of computations to hide the latency of long running operations. Message passing approaches in general, but web workers in particular suffer from a high cost of communication.

Finally functional languages generally discourage mutation of state. Instead, computations typically produce a fresh state, which may then be safely shared. However, such an approach is also not a perfect fit due to the object-oriented nature of JavaScript and the performance necessity to mutate state.

In River Trail, we adopt a compromise: In the spirit of functional programming, our model allows spawned tasks immutable access to their parent's state thus ameliorating the need to pass messages or call by value. In contrast to a purely function design, siblings are free to allocate and mutate their local heap. However, they cannot communicate with each other. This approach ensures safety while keeping runtime overheads low.

It is important to note here that the parent thread is suspended while its children are running and it therefore cannot mutate its own local state. Therefore, from the childrens perspective, the global state does not change. However, once all children have completed and the parent thread resumes, it is free to mutate its own local state. Ultimately, during sequential execution, the entire heap is mutable, which corresponds to the current JavaScript heap model. We call this approach *temporal immutability*.

2.2 API

River Trail provides a data-parallel API using the concept of parallel array as a fundamental abstraction of parallel computation. The API is built out of three components: a new data type *ParallelArray*, a set of *parallel methods* that specify the computation pattern and the concept of an *elemental function* that specifies the computation performed on each parallel array element. Figure 1 shows a simple example of a River Trail computation. A pair-wise addition is performed by calling the *map* method. It takes as an argument an anonymous elemental function that computes a sum of two values.

The ParallelArray type features a minimalistic API summarized in Table 1. ParallelArray objects consist of scalar values (single or double-precision floating point values or integers). ParallelArray objects can be created either from existing array-like data structures (such as, for example, JavaScript Array objects, another ParallelArray or HTML5 typed arrays) or by using the comprehension constructor, as illustrated in lines 2-3 of Figure 2. Here, the ParallelArray $ones$ is computed as a vector of length $a.length$ containing the values returned by the elemental function passed as a second argument of the constructor — in this case all ones.

A ParallelArray's elements can be processed using one of the six fundamental methods: *map*, *combine*, *reduce*, *scan*, *filter* and *scatter*. Map, reduce, scan and filter methods have standard data-parallel semantics. Combine is similar to map, except that it exposes the current index to the elemental function. The *scatter* method distributes elements from the ParallelArray into a new ParallelArray according to a given sequence of indices (akin to NESL's *permute* operation). River Trail's scatter, unlike the classical version, is mostly deterministic. In the situation when multiple values scatter to the same memory location, they are combined using a special conflict resolution function, which is passed to scatter as an argument. In case of a conflict with no conflict resolution function specified, scatter throws an exception. This guarantees deterministic execution of scatter, provided the conflict resolution function is commutative and associative. Because of this additional functionality, scatter effectively acts like a reduce from the map-reduce programming paradigm [9]. This usage scenario is illustrated in Figure 2, where a histogram of a ParallelArray's elements is computed by applying scatter to an array of all ones with addition as the conflict resolution function.

Constructors:

Signature	Creates
ParallelArray();	An empty ParallelArray
ParallelArray(arr);	A ParallelArray from an *Array-like* object arr
ParallelArray(pa_1, pa_2, ..pa_n);	A ParallelArray object of shape $\langle n, shape(pa_1)\rangle$
ParallelArray(s, f, ...$args$);	A ParallelArray of length s from a comprehension of f
ParallelArray($[s_0, s_1, ..]$, f, ...$args$);	A ParallelArray of shape $\langle s_0, s_1, ..\rangle$ from a comprehension of f

Methods:

Name	Signature	Elemental Function
Map	map(f, ...$args^*$)	$f(p_0$, ...$args)$
Combine	combine(s^*, f, ...$args^*$)	$f(index$, ...$args)$
Reduce	reduce(f, ...$args^*$)	$f(p_0, p_1$, ...$args)$
Scan	scan(f, ...$args^*$)	$f(p_0, p_1$, ...$args)$
Scatter	scatter($indices, defaultvalue^*, f^*, length^*$)	$f(p_0, p_1)$
Filter	filter(f, ...$args^*$)	$f(index$, ...$args)$
Flatten	flatten()	
Partition	partition(s)	

Table 1: The ParallelArray API. A * indicates optional arguments. f denotes a function object, s and s_i are scalars, pa_i is a ParallelArray object and p_i is either a scalar or a ParallelArray object. $shape(pa_i)$ returns a *shape* vector denoted by $\langle\rangle$ that describes the number of elements in each dimension of pa_i. A shape vector is always flat i.e., its elements are scalar values. For a detailed description of the API we refer the reader to the language specification at [14].

ParallelArray objects are immutable, i.e., once they have been created, their values can no longer be changed. Instead, operations on ParallelArray objects return a freshly minted ParallelArray (except for reduce, which returns a scalar). This restriction fits well with the overall programming model and allows us to perform optimizations, such as using a more efficient flat storage layout.

Note that our API does not feature many methods commonly found in other data-parallel languages, such as *add*, *sum*, *prefixSum* or *gather*. This minimalistic approach allows us to minimize the size of the compiler implementation, focus the language's design on the fundamental issues and, consequently, minimize the length of the re-design development cycle. Other methods can be easily implemented on top of our API as a JavaScript library. For example, *sum* can be implemented via reduce, while *gather* can be implemented via the comprehension constructor. In principle, we could have gone even further, and eliminate, map and combine, as both of these methods can also be expressed by means of the comprehension constructor. We include them nonetheless, as programmers expect built in map operations.

Elemental functions are essentially arbitrary JavaScript functions with the restriction that they may not mutate global state. Elemental functions are, thus, side-effect free. Elemental functions may mutate local state and access shared global state in a read-only fashion. It is the responsibility of the implementation to detect, possibly conservatively, a global state mutation and reject the execution of the corresponding ParallelArray method by throwing an exception. As elemental functions are side-effect free, the exception can be thrown at any point of the ParallelArray method execution cycle. This gives the implementation the flexibility to detect violations during either just-in-time compilation or at run time.

River Trail's programming model guarantees deterministic execution of map, combine and filter methods, equivalent to any sequential execution of these methods. This fact is a direct consequence of elemental functions being side-effect free. Reduce, scan and scatter are deterministic as long as their elemental or conflict resolution functions are commutative and associative. Otherwise, their execution is equivalent to some sequential execution. This trade-off is similar to that of the map-reduce programming model and seems to be well accepted by programmers.

While requiring both commutativity and associativity for guaranteed deterministic execution seems restricitive from a programmers perspective, it imposes the least constraints on concurrent implementations of reduce, scan and scatter. Such flexibility is particularly beneficial during prototyping and design space exploration. Yet, the current semantics also allow for refining the API in future versions by, e.g., lowering the requirements to just associativity, without breaking exisiting code,

River Trail naturally supports nested ParallelArray objects, i.e., ParallelArray objects whose elements are also ParallelArrays. It also supports generic n-dimensional programming inspired by languages like APL [16]. A Parallel-

```
1 function add(a, b) {
2   return a.map(function(e1, e2) {return e1 + e2;},
3                b);
4 }
```

Figure 1: Pair-wise addition in River Trail.

```
1 function histogram(a) {
2   var ones = new ParallelArray(a.length,
3     function(i) {return 1;});
4
5   return ones.scatter(a, 0,
6     function(e1, e2) {return e1 + e2;});
7 }
```

Figure 2: Histogram in River Trail.

Array object may encapsulate multiple dimensions and we consistently use index vectors instead of scalar indices in our API. As selection in JavaScript is based on scalars, we have added the *get* method that implements vector based selection.

Unlike a nested ParallelArray object, a multi-dimensional ParallelArray object is restricted in its shape: For each dimension, all elements have to have the same length. This allows us to encode a ParallelArray object's shape in a single vector of extents, available to the programmer via the *getShape* method. *flatten* and *partition* methods serve to change the dimensionality of multi-dimensional ParallelArray objects.

Figure 3 shows a straight-forward implementation of matrix matrix multiplication written using two-dimensional ParallelArray objects. The example uses a two-dimensional combine operator to compute each element of the resulting matrix in parallel, as implemented by the elemental function *matMultElement*. Line 12 shows the corresponding method call. The first argument to combine is the number of dimensions to iterate — in this case two, so that the elements to consider will be scalar values. Note that the elemental function, defined on Line 1, expects two arguments: the index and the second array *B*. A typical JavaScript program would simply use the closure bound *B* from the surrounding scope instead. However, due to the restrictions of our prototype implementation, we cannot support closure bound variables in elemental functions. JavaScript does not provide a reflection API to access the bindings of a function's closure. While this is a reasonable design choice in general, e.g., to ensure encapsulation, it prevents our prototype compiler, which is written in JavaScript, from reasoning about closure bound variables. A deeper embedding into the JavaScript runtime and just-in-time compiler would certainly provide access to closures but at the cost of a more complex implementation. Instead, we have chosen to adapt our API in the prototype and use the notion of *extra arguments*: All arguments following the second one in combine are directly passed through to the elemental function for iteration. We expect a product-quality implementation to use an API tailored towards closure bound variables, instead.

```
1 function matMultElement(index, B) {
2   var i = index[0]; var j = index[1];
3   var sum = 0; var len = this.getShape()[1];
4
5   for(var k = 0; k < len; k++) {
6     sum += this.get([i, k]) * B.get([k, j]);
7   }
8   return sum;
9 }
10
11 function MatrixMultiply(A, B) {
12   return A.combine(2, matMultElement, B);
13 }
```

Figure 3: Naïve implementation of Matrix Multiply in River Trail.

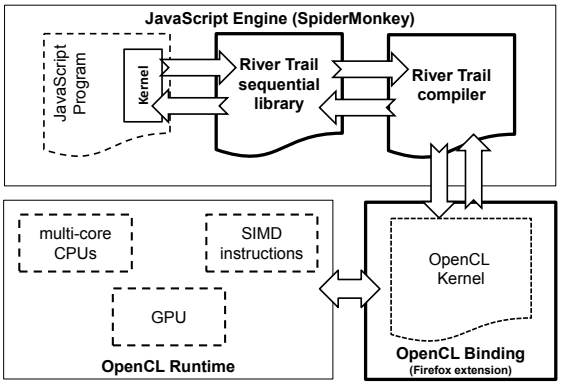

Figure 4: Design of the prototype implementation (River Trail components in bold)

3. Implementation

Our prototype consists of three major building blocks as depicted in Figure 4. First, we have written a JavaScript library that implements the ParallelArray API on top of JavaScript. Additionally, we implemented support for parallel execution for the Firefox browser, consisting of a compiler from JavaScript to OpenCL [21] and an OpenCL binding for Firefox.

Our library implementation is portable across modern browser engines and provides sequential execution for code using the River Trail API. Much care has been taken to implement the same semantics as in the compiled parallel mode. However, to make the use of River Trail practical in sequential code, we have prioritized performance where need be.

The sequential implementation is based on JavaScript's typed arrays — C like one-dimensional arrays that are laid out continuously in memory. By default the sequential library uses `Float64Array` arrays, but programmer annotations can specify other array types such as `Float32Array`. This enables direct interfacing with existing HTML5 APIs

like WebGL or canvas and also simplifies the implementation of our parallel runtime (cf. Section 3.2). As typed arrays are one-dimensional, we had to implement nesting and n-dimensional indexing on top of the flat data storage. We do this by wrapping the typed array object into our own ParallelArray object and implementing the index conversions in the *get* method. To reduce runtime overheads, we dynamically specialize the implementation of get to the dimensionality of the array. We also implemented regular JavaScript indexing with `[]` operations by intercepting property lookup using JavaScript's proposed *proxy* feature. However, runtime overheads turned out to be intolerable. Therefore, the library, unlike the compiler, does not support direct indexing.

The sequential library also implements all parallel methods, albeit in sequential versions. We do not check for side effects of elemental functions in the library implementation, nor do any of the limitations of the compiler apply. However, we do enforce homogeneity and shape constraints on the computed ParallelArray objects. If a result does not comply, we gracefully fall back to nested JavaScript arrays to enable debugging. However, performance significantly suffers in those cases.

Parallel execution is enabled by our compiler. It is invoked by the sequential implementation in case of the comprehension constructor and the map and combine methods. When such a method is called, we dynamically check whether the compiler is present and was successfully initialized. If both conditions are met, we pass control to the compiler which translates the elemental function into an OpenCL kernel function. On successful compilation, the generated kernel is executed using the OpenCL runtime binding. We then transform the result into a new ParallelArray object and resume sequential execution. On failure or if the compiler is not present, we fall back to the sequential implementation.

Using OpenCL as the underlying runtime and compilation backend allows us to target different hardware platforms, like multi-core CPUs and programmable GPUs, from various vendors without investing into dedicated compilation support for each target. While this significantly eases the implementation of a portable runtime, it also comes with a range of challenges.

Firstly, JavaScript and OpenCL differ significantly from a language perspective. Whereas the former is a dynamically typed language aimed at designers and beginner programmers that abstracts from hardware specifics as much as possible, the latter is a statically typed language from the C family of languages designed for performance experts and exposes many hardware specifics. A major part of our implementation therefore is concerned with bridging this semantic gap from dynamic to static and hardware agnostic to hardware specific.

Secondly, River Trail employs a restricted shared memory programming model where all threads during concurrent execution have restricted read-only access to the global heap. OpenCL on the other hand is based upon a distributed memory model where data is only accessible during concurrent execution if it has previously been explicitly mapped. Thus, our implementation has to infer the correct mapping and manage the communication.

Lastly, OpenCL and its runtime use explicit memory management in the spirit of C, where all data structures are explicitly allocated and, more importantly, have to be explicitly freed. JavaScript, on the other hand, uses a fully managed runtime and employs garbage collection to free unused resources. This poses two problems: In code that we translate to OpenCL, we have to convert JavaScript's implicit allocations to explicit allocations in OpenCL. This is complicated further by the lack of per-thread heaps in OpenCL. Another challenge in this context is to ensure that the lifetime of memory allocated in OpenCL covers what is required by the semantics of the translated code. The second issue arises in the implementation of the actual runtime. As the lifetime of heap allocated objects in JavaScript is ultimately determined by the garbage collector, we have to keep the state of the OpenCL runtime alive until the garbage collector of the JavaScript runtime signals that it is no longer needed. OpenCL runtimes are typically not optimized for the resulting deallocation patterns, leading to intolerable garbage collection latencies. The design of our embedding of OpenCL into the Firefox browser takes this into account.

In the following, we describe our solutions to the above challenges and discuss the design decisions involved.

3.1 Translating JavaScript to OpenCL

We have not aimed at writing a general purpose compiler for JavaScript to C like languages. Instead, the development of our compiler was largely driven by use cases and the associated developer feedback. It is geared towards numerical codes and only supports those language features that are typically required in that setting. In particular, our prototype does not support

Closure bound variables As discussed earlier, supporting closures would require a deeper integration into the JavaScript engine.

User thrown exceptions Using *throw* triggers a fall back to the sequential implementation. For exceptions thrown by the JavaScript runtime or supported library functions, we ensure that concurrent execution is aborted and switch to sequential execution to produce the actual exception.

Objects Exceptions are array objects and special objects like the *Math* object whose methods provide essential arithmetic operations required for numerical workloads. We also support the use of objects as records in limited contexts.

Value polymorphism In order to be accepted by our prototype compiler, variables need to refer to data of the same

type throughout their lifetime. This is typically the case in numerical workloads. Note, however, that functions may be used polymorphically.

Strings We did not encounter a use case for strings in our benchmark applications. As strings typically use implementations in today's jit engine that trigger internal side effects, we have deferred an implementation until the need arises.

These restrictions aside, we have taken great care to ensure that the semantics of the compiled code is in line with JavaScript's semantics.

The majority of restrictions mentioned above are due to our prototype implementation and not a property of the programming model itself. By embedding the compiler deeper into the JavaScript engine, most of the above functionality can be implemented. A notable exception is objects, in particular non-native objects like DOM nodes. As their semantics are external to JavaScript, it is difficult to reason about their side effects or even dynamically capture their behaviour. Native JavaScript objects, in contrast, have known semantics and could be analyzed by the JavaScript compiler. We will further discuss above limitations and their respective reasons inline with the following description of the compiler.

The compiler itself is written in JavaScript and runs alongside the programmer provided scripts. This brings two benefits: Firstly, the compiler is portable across different browser engines and thus only the runtime has to be adapted. Secondly, embedding the compiler in the existing JavaScript context gives us an easy way to intercept calls to ParallelArray methods and divert them to OpenCL execution instead.

In principle, this would also work the other way round: a malicious web site could intercept calls to the runtime and inject some exploit code. This problem, however, is not unique to the work presented here and in particular Firefox, which itself is partially implemented in JavaScript, offers means to prevent such attacks [10]. Although possible, hardening our prototype compiler using those techniques is outside the scope of this work.

Figure 5 gives an overview of the compiler stages. First, as most compilers, we parse the elemental function into a syntax tree using the parser from Mozilla's Narcissus metacircular JavaScript interpreter[3]. We use JavaScript's ability to reflect the source of any function by calling the function object's `toString` method. Second, we annotate the syntax tree with types. For this, we have implemented an inference for a simple first-order type language with size information. We reject programs that make polymorphic use of variables and resolve function polymorphism by specialization. This enables us to produce efficient OpenCL code without the need for tagged data representations. We discuss details in Section 3.1.1.

[3] Narcissus is available at `https://github.com/mozilla/narcissus`

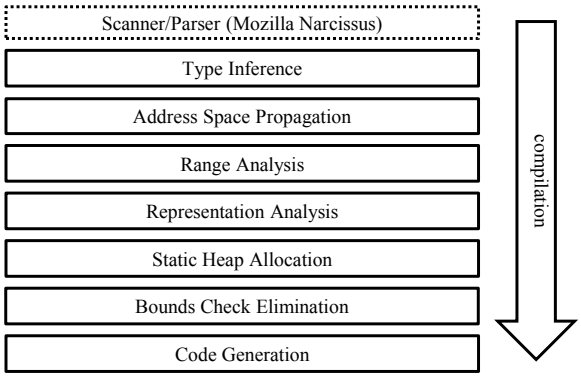

Figure 5: Overview of compilation stages of our prototype compiler

The *type inference* infers types only up to the JavaScript level. It does not take OpenCL's address spaces nor the different number types in OpenCL into account. This gap is closed in the following two stages.

During *Address Space Propagation* we compute address spaces for all local variables. The parameters and return value of an elemental function are always allocated in the *global* address space, as they need to be accessible from the JavaScript environment. All other variables are by default allocated in the *private* address space. However, to emit valid OpenCL code, we have to prevent data from the global heap to be referenced by variables from the private address space and vice versa. To that effect, we forward propagate the global address space along the data-flow graph, promoting variables to the global address space where needed. Rare conflicts where both private and global data flows to the same local variable are resolved by copying the global data into the private heap. Address spaces are computed for the whole kernel including local functions. Consequently, updates in address spaces may trigger further function specialization.

Beyond code generation, we also use the computed address space partitioning to statically check for side effects due to array updates. We enforce the constraint that only local arrays may be mutated by checking that the updated array was allocated in the *private* address space. This condition ensures that no globally visible side effects are performed. In the rare case that a global value was privatized to resolve an address space conflict, we might allow an update that would otherwise be illegal. However, due to privatization, the generated code is still side effect free.

In a second stage, we introduce OpenCL specific types. It often is beneficial to differentiate between Integers and floating point values at runtime. Most JavaScript engines implement support for 32 bit integers to that effect. To decide whether a value would overflow a 32 bit Integer at runtime, we next perform a range analysis (cf. Section 3.1.2). Apart from computing ranges for all local variables, the analysis

```
__kernel void RT_matMultElement(__global double* opThis,
                                __global double* RTl_B,
                                __global double* retVal)
{
  int _id_0 = get_global_id(0);
  int _id_1 = get_global_id(1);
  int _writeoffset = _id_1+1024*_id_0;
  double RTl_sum;
  int RTl_len;
  int RTl_k;

  RTl_sum = (double) 0;
  RTl_len = 1024;
  for (RTl_k = 0; (RTl_k<RTl_len); RTl_k++) {
    RTl_sum += opThis[RTl_k + 1024*id_0]
               * RTl_B[id_1 + 1024*RTl_k];
  }
  retVal[_writeoffset] = RTl_sum;
  return;
}
```

Figure 6: Pseudo OpenCL code generated for Matrix Multiply when applied to a 1024×1024 matrix.

also infers whether a variable is known to be integral. The representation analysis stage then uses the information to decide a global representation for each variable.

Next, we perform static memory allocation. OpenCL does not provide thread local heaps nor allow dynamic allocation on the stack. Instead, only objects of statically known size may be allocated using C99 style local arrays. Based on the size information computed by type inference, we statically compute allocation sets. We use a path aware analysis to reduce the pressure on the stack, i.e., we overlap allocations for different branches of conditionals.

As a last optimization, we eliminate bounds checks. In JavaScript, all array accesses need to be bounds checked. Our current implementation naïvely inserts two conditionals for each array access. However, as we have precise size information available from our type inference and also approximations of variable ranges, eliminating bounds checks is straight-forward.

Finally, we emit OpenCL code. Figure 6 gives pseudo code for a compilation of the Matrix Multiply example from Figure 3 assuming a 1024×1024 matrix as input. As can be seen, the code is structurally very similar to its JavaScript counterparts. However, arrays are represented as pointers of appropriate type. Note in particular the use of an Integer induction variable and the absence of bounds checks in the inner loop.

Before discussing our OpenCL runtime binding, we first give more details on type and range analysis.

3.1.1 From Untyped to Typed

We use a straight-forward first order, monomorphic type system. This allows us to directly map the untyped JavaScript program to OpenCL without the need for boxed data representations and value polymorphic code. Although JavaScript

$$
\begin{array}{lcl}
p & \Rightarrow & \texttt{number} \mid \texttt{bool} \mid \texttt{string} \\
a & \Rightarrow & \texttt{array(} \left(\mathbf{a} \mid \mathbf{p} \right) \texttt{,} \ \mathbf{n} \texttt{)} \\
 & \mid & \texttt{parray(} \left(\mathbf{a} \mid \mathbf{p} \right) \texttt{,} \ \texttt{[} \ \mathbf{n} \left[\texttt{,} \ \mathbf{n} \right]^{*} \texttt{]} \texttt{)} \\
o & \Rightarrow & \texttt{object(} \left[\texttt{(} \ \mathbf{L} \ \texttt{,} \left(\mathbf{a} \mid \mathbf{p} \right) \texttt{)} \right]^{*} \texttt{)} \\
f & \Rightarrow & \texttt{function(} \left(\mathbf{p} \mid \mathbf{a} \mid \mathbf{o} \right) \left[\texttt{,} \left(\mathbf{p} \mid \mathbf{a} \mid \mathbf{o} \right) \right]^{*} \texttt{)}
\end{array}
$$

Figure 7: Type language used for type inference. $\mathbf{L}$ is a set of labels and $\mathbf{n}$ represents natural numbers.

code in general has been found to be very dynamic [27], this seems not apply in the restricted setting of elemental functions. In our experience, elemental functions focus on computationally dense numerical operations, which do not require polymorphism. Furthermore, our prototype only has limited support for method invocations, which is a significant source of polymorphism found in general JavaScript code. Overall, these restrictions had a surpising little impact on the actual JavaScript workloads we can compile.

We also include size information in our types, thereby effectively performing a form of shape analysis at the same time. Having precise shape information available allows us to compute the size of all heap objects and thereby enables static allocation.

We omit a formal discussion of the type system, as it closely follows text book approaches [24]. To nonetheless give the reader a flavor, we present the type language in Figure 7. p denotes the set of types for *primitives* as supported by JavaScript. Note that, although our type language supports the `string` primitive, we do not have implemented support for it in the backend. We also do not support the undefined or null value, thereby ensuring that each expression produces an actual value.

We represent arrays by two flavors of array types a. The first, `array`, is used for ordinary JavaScript arrays as well as typed arrays. It has two parameters: the type of the elements of the array and the length of the array. As this shows, we require arrays to be homogeneous, i.e., all their elements need to be of the same type, and their length needs to be statically known. We allow ordinary JavaScript arrays to be nested, i.e., the elements of such an array may be other arrays.

Our second array type, `parray`, covers ParallelArray objects. It differs from the ordinary array type in that the length parameter is a vector of numbers to be able to model multi-dimensional ParallelArray objects. The main reason to use two separate types lies in their different APIs. To correctly resolve method calls, we need to know what kind of array we are dispatching for.

Next, production rule o of our type language introduces object types. For simplicity, we do not support objects beyond the two array types mentioned above. However, we include a limited form of nameless structural objects to support a JavaScript pattern commonly used to return multiple values:

```
1 function foo() {
2   return {desc: "answer", value: 42};
3 }
4 var answer = foo().value;
```

To be able to return two results from function `foo`, we construct a nameless object in Line 2 that we use as a set of name-value pairs. The calling context can then extract the different result values using their name, as shown in Line 4.

Lastly, we have a function type. In our system, all functions have to return a value, represented by the first parameter, and may accept an arbitrary number of arguments. Note here that we, unlike JavaScript, require the number of function parameters and arguments in a call to match. This limitation, which has little impact on expressiveness, ensures that no undefined values creep into our supported subset of JavaScript.

For function types, monomorphism makes a difference. Whereas JavaScript developers write monomorphic code when it comes to values, they often write rather polymorphic functions. Having two array types further increases the amount of polymorphism: Programmers typically write helper functions on arrays only once. We solve this issue by aggressive specialization for different primitive types, different array kinds (parallel vs. ordinary) and even different array sizes. Whilst this potentially could lead to dramatic code growth, our experience shows that the amount of specialization remains manageable. In the codes we studied, typically only few different array sizes are used. Specializing for the two kinds of arrays can maximally double code size but leads to less code duplication in practice.

3.1.2 Lightweight Range Analysis

We have implemented an abstract interpretation phase that computes the lower and upper bounds for each expression and whether it is guaranteed to evaluate to an integer value at runtime. To support generic n-dimensional codes, our analysis computes vector bounds for vector typed variables. Certain expressions, like the index parameter of an elemental function or integer constants, are known to be integer values and seed the analysis. Bounds are propagated across a set of arithmetic operations and derived from conditionals in the usual way. Predicate expressions in loops, however, are treated special.

To compute range information for loops, typically a trip count is required to project the range of variables after the loop exits. However, given our time constraints both in compiler runtime and development, we have opted not to implement loop analysis. Instead we use abstract interpretation directly. Consider the following loop:

```
1 var i = 0; var s = 0;
2 while (i <= 10) {
3   s = s + A[i++];
4 }
```

When evaluating the above code, we treat the loop predicate `i <= 10` not as a constraint like we do with ordinary conditionals. Instead, we interpret it as upper bound for the variable `i`. Thus, in the loop body, `i` has lower bound `0` derived from the assignment in Line 1 and upper bound `10` derived from the loop predicate. For the above example, continuing the analysis with this information gives us the correct bounds (`i` is assigned $[0, 10]$ and `s` is assigned the unknown bounds). However, in general, the loop predicate might not be good enough to correctly predict all ranges if the loop has further induction variables that are only indirectly constrained. Below is an example:

```
1 var i = 0; var j = 0; var s = 0;
2 while (i <= 10) {
3   s = s + A[i++] * B[j++];
4 }
```

After the first round of analysis, `i` is annotated with the correct range as before, whereas `j` has range $[0, 1]$, which is incorrect. To detect these cases, we rerun the analysis once more, taking the previous results into account. After the second run, `i` still has the range $[0, 10]$, as its upper bound is constrained by the loop predicate. The range for `j` however has changed to $[0, 2]$, as the variable is not constrained. If we detect such a situation, we invalidate the range information of the corresponding variables and propagate an undefined range instead.

As our analysis is monotonic, two iterations are sufficient to detect variables with unconstrained ranges. If a bound increased in the second iteration compared to the first, it will further increase in the third iteration. Likewise, if a variable's range did not change in the second iteration, it will not change in the third either. Furthermore, our inferred ranges are accurate, *i.e.*, if we are able to compute a variable's range, the computed range is always a superset of the observed range at runtime. Although our analysis only covers a subset of the loops that a full loop analysis would handle, we have found it to be sufficient for the workloads we have considered.

3.2 Integrating the OpenCL runtime with Firefox

All components of the prototype that we have described so far are written completely in JavaScript and run fully inside the browser engine. However, to actually execute the OpenCL kernel code that our compiler has generated, we have to leave the browser and employ the help of an OpenCL runtime. For this task we have implemented a lightweight embedding of the OpenCL runtime into Firefox's JavaScript engine. Other than the rest of the prototype, this component is written in C++. We use Firefox's XPCOM binary extension mechanism to load our extension into release versions of Firefox and the JSAPI interface to SpiderMonkey, Mozilla's JavaScript engine, to communicate with the JavaScript world. Our implementation does not aim to provide a full interface to OpenCL. Instead, we have tailored the API to the needs of our River Trail implementation.

3.2.1 Runtime Optimizations

Our runtime implements a range of common optimizations to reduce JIT and OpenCL runtime overheads. To increase the portability of our runtime, we have aimed to implement as much of the required logic in JavaScript and keep the C++ component as small as possible.

To reduce the cost of just-in-time compilation, we cache compiled kernels as long as the corresponding JavaScript function object is alive. We realize this by exposing compiled kernels as JavaScript objects in our C++ interface and attaching those kernel objects to the respective JavaScript function object. By making compiled kernels explicit in the runtime, we can reuse the existing garbage collection mechanisms of the JavaScript engine. During finalization of a JavaScript kernel object, we also free the corresponding OpenCL data structure.

We use a similar approach to manage data transfers and OpenCL heap mapping. Before data can be used by an OpenCL kernel, it has to be explicitly mapped. Even though hardware trends towards shared memory systems, mapping data still imposes noticeable overhead. Therefore, a common runtime optimization is to reduce transfers where possible. We, too, have implemented a form of *lazy materialization*. As before, the key technique is to expose mapped memory as explicit JavaScript objects. This allows us to cache mapped data by attaching the object that represents the mapping to the JavaScript object that has been mapped. Again, lifetime management is performed by the existing JavaScript garbage collection. Note that we only cache ParallelArray objects, as these are immutable and therefore guaranteed to remain consistent with their mapped copy during their lifetime. Extending our approach to mutable objects would require some form of consistency protocol that forwards changes to an object from the JavaScript heap to the OpenCL heap's counterpart.

Our lazy materialization goes one step further: For kernel results, we do not map the result back to the JavaScript heap unless the value is actually read. We implement this in JavaScript by replacing the data store of a ParallelArray object with the object that represents the corresponding OpenCL memory buffer. To ensure that the data is mapped on read, we dynamically override all reading methods with special implementations that first map the data back to the JavaScript heap. Our implementation exploits the dynamic nature of JavaScript, in particular the ability to add new methods to an object at runtime, thereby overriding methods inherited from the object's prototype. To keep overheads low, we reset those methods to their default implementation as soon as the data has been mapped.

Reusing JavaScript's existing garbage collection mechanisms to manage the lifetime of OpenCL runtime objects has proven to be an efficient implementation strategy. However, a too naïve implementation may show unexpected runtime costs caused by the interplay of garbage collection and OpenCL runtimes.

The OpenCL runtime, unlike JavaScript, uses explicit memory management with reference counting. Under that regime, memory is reclaimed as soon as the last reference to it is surrendered. Consequently, an OpenCL runtime is typically tuned for single free operations that are intertwined with overall program execution. Garbage collected runtimes, on the other hand, free many objects at once whenever a collection cycle is completed. In our prototype, this includes the OpenCL runtime objects and their JavaScript representations. The resulting bursts of free requests result in significant overheads of up to 200ms. In real time centric use cases like video processing or 3D animations, such delays are very noticeable.

To alleviate the effect, we have decoupled freeing the wrapper objects from releasing the OpenCL runtime objects. Instead, we now maintain a global free queue that contains OpenCL runtime objects that are no longer needed. During garbage collection, instead of calling the OpenCL API function to release an object, we simply add it to the free queue. During certain points of execution, in particular all situations where new OpenCL runtime objects are allocated, we inspect the queue and release a small amount of pending objects. Using this approach, we were able to reduce garbage collection latency to 20ms, without otherwise noticeable effect on overall performance.

Last but not least, we have implemented memory alignment: In particular for vectorization on CPUs, data typically has to be aligned in memory. OpenCL runtimes report this alignment requirement at runtime. This allows implementations with explicit memory management to allocate heap structures accordingly, typically by allocation a larger chunk of memory and computing an offset to gain an aligned start address. However, our shallow embedding does not allow us to directly influence the allocator of the used JavaScript runtime.

Instead, we exploit a feature of the typed array specification: Typed arrays in JavaScript are defined as views over a byte buffer, which can be allocated independently. In particular, the view may begin at an offset in the underlying buffer. The last missing ingredient is a way to compute this offset. This cannot be done in JavaScript, as JavaScript has no notion of pointers and does not expose the address of objects to the program. We have added this missing pointer reflection mechanism in our C++ component.

3.2.2 Hybrid Execution

Using OpenCL as backend technology allows us to transparently run River Trail workloads on the CPU and GPU. Even more, we can also execute workloads on both devices at the same time. To evaluate the benefits of such a hybrid approach, we have implemented a simple static scheduler. The basic idea is to map all read-only inputs, i.e., all kernel parameters, to both the CPU and GPU devices. The work-

load distribution is determined by an offload factor specified at runtime that determines the portion of the iteration space that should be computed on each device. After this partitioning, computation proceeds as usual with each of the CPU and GPU devices producing results into their own regions of the result buffers. After completion of execution on all devices we map all sub buffers back to the host and return the combined result.

4. Experimental Evaluation

We present an experimental evaluation of the River Trail prototype compiler using several realistic web applications. Specifically, we focus on the following aspects:

1. **Performance and scaling on a modern CPU** relative to sequential JavaScript.
2. **Performance impact of optimizations**, specifically dynamic bounds check elimination and lazy materialization.
3. **Performance on a modern integrated graphics unit** relative to sequential JavaScript with and without hybrid execution.

All experiments were run using Firefox 16 on a machine with an Intel Core i7-3770 CPU with 4 cores (8 hardware threads) clocked at 3.4GHz and with 4GB of memory. This machine also contains an Intel HD Graphics 4000 integrated GPU with 16 execution units (each running 8 threads, for a total of 128 threads) clocked at 650MHz and with 1.7GB of shared memory.

4.1 Workload Selection

River Trail is a disruptive technology and as such no suitable workloads exist. Today's benchmarks like Sun Spider or the V8 benchmark suite [30] focus on sequential performance and are not representative of real web sites [26]. Even more, current web applications are designed to run within the computational envelope provided by today's JavaScript engines and thus are not very computationally intense.

To escape this chicken-and-egg problem, we have implemented and sourced a range of workloads, summarized in Table 2. Of these, 5 were developed by third parties who first developed sequential versions and then ported them to the River Trail API. These third parties included web designers, professional developers, academic researchers as well as undergraduate students. We believe that the set of applications we have chosen is representative of emerging HTML5 applications and allows us a fair evaluation of River Trail's potential. Below, we briefly describe each of these workloads and relate them to the performance speedups and scalability shown in Figure 8.

4.2 Parallel Speedup

The chart in Figure 8 shows the impact of the number of hardware threads on parallel speedups observed. The Y-axis is the ratio of time taken for sequential execution and time taken for River Trail execution. Note that the processor uses hyper threading. Thus, bars labeled 2, 4, 6 and 8 hardware threads correspond to runs on 1, 2, 3 and 4 cores, respectively, with two hardware threads each. The bar labeled as 1 hardware thread, however, runs an exclusive hardware thread on a single core. We attribute the reduction in speedup for 8 of 11 programs on two hardware threads compared to the execution on a single hardware thread to hyper threading and runtime effects. We rely on the OpenCL runtime to spawn the right number of *user* threads and decide an efficient work assignment. We therefore do not explicitly control nor know the precise concurrency during execution. Aside from scalability this chart also implicitly shows the contributions of parallelism and code generation to parallel speedup.

The second chart in Figure 8 shows the impact of the optimizations described in Section 3. All bars show the relative speedup compared to a fully optimized version with all optimizations described in Section 3 turned on - in the discussion below, we refer to this optimized version as AllOpts. The bars labeled NoLazyMaterialization show relative speedups when lazy materialization is turned off but all the other optimizations are on. The bars labeled NoDynamicBoundsChecks show relative speedups when no dynamic bounds checks are emitted. These bars therefore illustrate the performance speedup that can be achieved with an ideal Range Analysis. Note that this artificial scenario is used only for evaluating effectiveness of our RangeAnalysis and is not an optimization itself. The bars labeled FullDynamicBoundsChecks show relative speedups with all array selection operations guarded with checks. These bars therefore show speedup without any static Range Analysis.

TourDeBlock is a game application that features a *rigid body physics engine* to simulate realistic collisions and movement. The collision detection phase of the program accounts for 50-60% of total execution time per frame. This phase consists of two sub-phases *Broadphase* and *Narrowphase* each of which are made parallel using River Trail. We observe a parallel speedup of around 2.84 (Figure 8a) for the collision detection phase alone and 1.6 for the entire application. The elemental functions for this workload contain dense irregular control flow and as a result they do not benefit from auto-vectorization. This program also does not benefit from lazy materialization as the two parallel subphases are separated by a sequential computation that reads the results of the first sub-phase.

XML3D simulates an interactive walk-through of a virtual museum in which the user views specific art installations. The scene consists of several human characters that are made of high-detail meshes and a bone and joint model for realistic movement. These meshes are *skinned* and *animated* in realtime, which is the most compute-intensive portion of this application and takes up on average roughly 50-70% of execution time per frame during sequential execution. With

Name	Description	SLOC	# kernels	Input Size	% parallelized
TourDeBlock	Real-time Physics engine	8824	2	250 bodies	55-60
XML3D	Interactive Virtual Museum	15000	2	Mesh sizes	50-70
VideoEditor†	Real-time filters on a video stream	1050	7	640x480 video stream	64-90
Matrix-Multiply†	Dense Matrix Multiplication	30	1	1024 x 1024	>99
Nbody	Particle simulation/animation	2035	2	4000 bodies	98
Liquid-resize†	Content Aware image resizing	862	3	800x542 image	85
Bugs	Interactive gestures based game	3300	10	640x480 video stream	97
OctreeCollider	Octree based rigid body collision engine	2600	1	10000 bodies	85

Table 2: Summary of applications. Applications marked with † were developed by the authors and the rest by third parties. SLOC refers to the total number of lines of JavaScript. % parallelized refers to the portion of execution time during sequential execution in components that were targeted for parallelization using River Trail

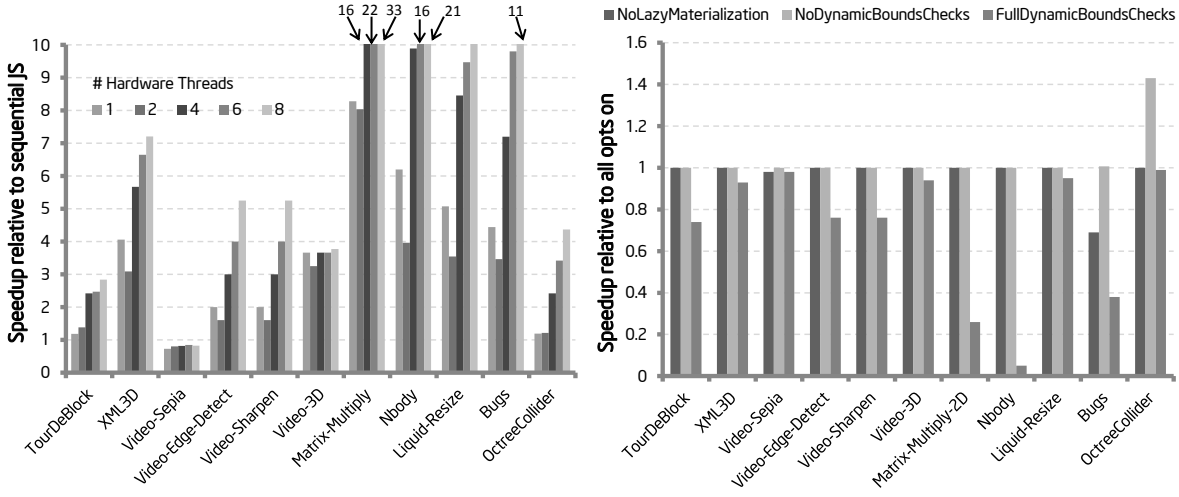

Figure 8: (a) Parallel speedup relative to plain sequential JavaScript versions and (b) the impact of optimizations on performance - the bars show speedup without specific optimizations relative to execution with all optimizations turned on (using 8 hardware threads in both cases)

these components parallelized using River Trail we measure a speedup of 7.21 (Figure 8a) for the parallelized portion and a speedup of roughly 2 for the whole application.

Video-* workloads are part of an in-browser video editor application in which users apply various effects to a video stream. Video-Sepia, Video-EdgeDetect and Video-Sharpen refer to the user applying *sepia toning*, *edge detection* and *image sharpening*, respectively. Video-3D refers to the input stream being transformed to *stereoscopic 3D* in real-time. We observe speedups ranging from 0.87 (for Video-Sepia) to 5.25 (for Video-EdgeDetect and Video-Sharpen). Lazy materialization does not provide any improvement since the results of the elemental functions are rendered to the screen immediately after parallel execution. However, the static range analysis is effective in eliminating bounds checks as seen by the slowdown with FullDynamicBoundsChecks in Figure 8b.

The slowdown for Video-Sepia in Figure 8a (in contrast with the significant speedups for the other Video-* programs) is a consequence of two factors. Firstly, the amount of work parallelized in Video-Sepia is relatively low, which emphasizes overheads. Secondly, sepia can be computed in place by the sequential implementation by directly updating pixel values of the original image buffer. River Trail's programming model does not allow for this. The other programs, however, cannot be performed in place due to loop-carried dependencies.

Matrix-Multiply workload implements the standard O(n^3) algorithm for dense matrix multiplication which is excerpted in Figure 1. We observe a speedup of 33.6 for this program (Figure 8a). Effective range analysis contributes significantly to this speedup — with FullDynamicBoundsChecks the speedup is about 8.8x over plain sequential JavaScript (i.e., a speedup of 0.26x over the optimized case which was 33x faster than sequential JavaScript). These dy-

namic bounds checks are also an impediment to vectorization which the All-Opts case benefits significantly from. For comparison we wrote a native version of this workload using C and OpenCL running on the CPU that uses the same O(n^3) algorithm. We observed that the River Trail program runs 0.97x as fast as this native OpenCL implementation on 1024 x 1024 matrices. For this program, our compiler is therefore able to generate code whose performance is on par with native code.

Nbody implements an O(n^2) simulation and animation of particles that are moving through a force field and interacting with each other gravitationally. Roughly 98% of the execution time in each frame is spent computing *positions* and *velocities* of the particles and these computations can be parallelized effectively resulting in a parallel speedup of 21.5 for these computations and a roughly equivalent improvement in overall frame rate (Figure 8a). The impact of our range analysis is particularly evident in this program — in the FullDynamicBoundsChecks case the parallel speedup is only 1.12 (Figure 8b). On the other hand speedup for the NoDynamicBoundsChecks is roughly equal to the speedup for All-Opts, i.e., our analysis is able to eliminate most of the performance critical bounds checks.

Liquid-Resize implements *content-aware image resizing* based on the algorithm in [3]. The portion of this program parallelized using River Trail accounts for roughly 85% of the total sequential execution time. The parallel speedup in just the parallelized portion of the program is approximately 10.9 (Figure 8a) and the overall application speedup is 2.2 over sequential JavaScript execution. The speedups are not significantly affected by lazy materialization.

Bugs is an interactive game in which players use gestures to interact with and move bugs in a scene. This application implements dense optical flow tracking using Farneback's method [12]. The parallel version of this method is implemented with a chain of 10 relatively short, vectorizable elemental functions that are invoked for each input frame. We observe parallel a speedup of about 11.8 for the All-Opts case. The chain of elemental functions produce and consume large ParallelArray objects without intervening accesses in sequential code. Therefore in this case lazy materialization has a substantial impact on parallel performance — without this optimization the speedup is roughly 4.5. Bounds check elimination also has a big impact on speedup — parallel speedup is improved by 2.6x (11.8x vs. 4.5x) compared to FullDynamicBoundsChecks.

OctreeCollider is a physics engine that simulates collision detection and dynamics on rigid meshes. Unlike TourDeBlock which uses a brute force $O(n^2)$ broadphase for detecting collisions, this application uses an octree-based scheme to efficiently prune out non-colliding objects. Parallel execution for this program is 4.3 times faster than sequential JavaScript and lazy materialization does not affect parallel performance significantly. However performance with NoDynamicBoundsChecks is significantly better than All-Opts. The sole elemental function contains a `while` loop whose termination condition is data-dependent. Moreover most array selection operations within this loop use indices that are also data-dependent. Range analysis is thus unable to resolve bounds precisely.

It is important to note here, that, beyond just speeding up applications, River Trail also acts as an enabler for new kinds of applications. For instance, the NBody program runs at 2 frames/second in sequential JavaScript. With River Trail it runs at around 43 frames per second — sufficiently fast for a near real-time experience. Similarly the Bugs game runs at less than 1 frame per second in sequential JavaScript. With River Trail, it runs at about 12 frames per second — sufficient performance for interactive play. The same is true for Video-EdgeDetect and Video-Sharpen: Each runs at around 4 frames/second sequentially whereas the parallel versions run at around 21 frames/second, which is closer to the original playback rate and is discernibly smoother.

Figure 10 shows the total cost in milliseconds of JIT-compiling all the elemental functions invoked for each workload during parallel execution. The elemental functions in Tour De Block and the OctreeCollider are both quite large and consist of deeply nested loops. This is reflected in the relatively large amount of time spent in Type Inference and the time taken for Range Analysis to converge. However the cost of JIT compilation is low relative to the application execution time for most of the studied programs. Applications that are particularly sensitive to the pauses due to JIT compilation (for example TourDeBlock) are able to pre-compile the elemental functions during page or application loading. We observed that re-compilation (or re-specialization) is rarely triggered (thereby avoiding "jitter" in an otherwise smooth animation or video rendering for example).

4.3 Hybrid Execution

The River Trail runtime supports execution of kernels on GPU devices transparently to the programmer and the compiler. The performance speedup due to hybrid execution for NBody and Matrix-Multiply-1D from Table 1 are shown in Figure 9 for various input sizes. The iteration space for a ParallelArray operation is partitioned between the CPU and GPU statically: CPUx-GPUy means that x% of the iteration space is computed on the CPU and y% on the GPU. CPU-only and GPU-only refer to computing the full iteration space on CPU only and GPU only, respectively.

The parallel performance of NBody for 4000 bodies increases as the portion of the iteration space offloaded to the GPU in increased up to 75%. With GPU-only execution the speedup is lower than with CPU25-GPU75. This can be attributed to the significant amount of rendering that this program performs on the GPU interleaved with computation. When the problem size is scaled up, the amount of rendering

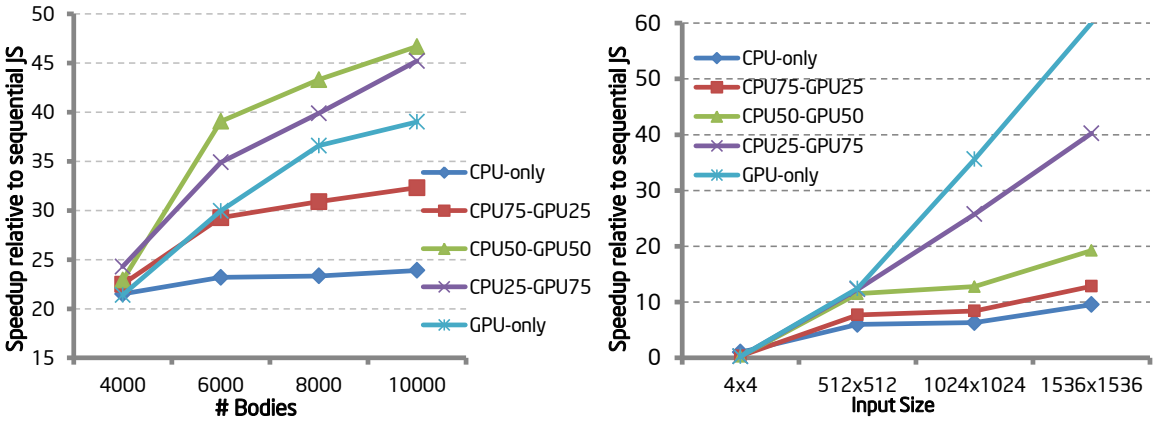

Figure 9: Parallel speedup for (a) NBody (on the left) and (b) Matrix-Multiply (on the right) with hybrid execution relative to sequential JavaScript execution. The Matrix-Multiply-1D program shown here uses a flattened 1D representation of the matrices and is distinct from the 2D version in Table 2.

performed on the GPU also increases. The best performance then is achieved instead with a 50-50% partitioning. We also observe that the performance scales better with problem size on the GPU-only and hybrid configurations than on the CPU-only configuration.

The plot on the right-hand side of Figure 9 shows the performance of hybrid execution for Matrix-Multiply. This program both performs better and scales better than the NBody workload. We attribute this to two main reasons: Unlike NBody, this program does not perform any rendering on the GPU and the elemental function contains no significant control flow divergence. The latter also makes it particularly suitable for GPU execution: GPU-only configuration produces the best speedups and performance decreases with increased CPU utilization.

These experiments indicate the large number of factors that influence the best partitioning for hybrid execution. In addition, there is the overhead of spawning elemental functions on a GPU device. To be effective, a hybrid execution or partitioning scheme has to balance these considerations carefully to produce good results across a variety of workloads. There has been a significant amount of exploration in this area by other researchers (see [25] and references therein).

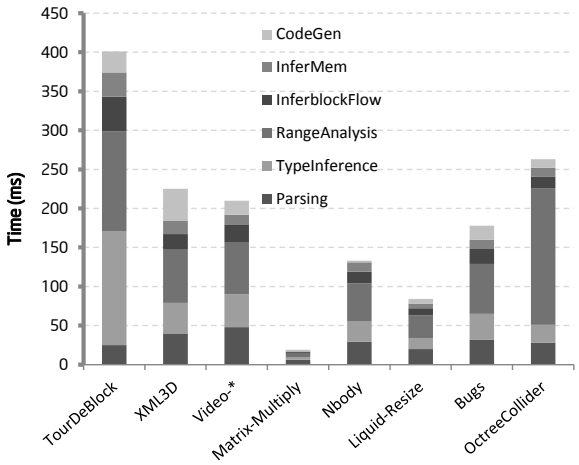

Figure 10: Breakdown of JIT compilation times for the workloads in Table 1.

5. Related Work

Data-parallel programming is a well understood concept. It has been extensively studied [5] and approaches have been developed for a wide range of languages. However, to our knowledge nobody so far has addressed JavaScript.

Most existing approaches (for example ArBB (C++) [22], DpH (Haskell) [8], Lime (JAVA) [11], Firepile (Scala) [23], Accelerator (.NET) [28], to name few) cover statically typed languages that are compiled ahead-of-time. ASDP [25] comes close by extending ActionScript, which is similar to JavaScript but has type annotations. In their DSL, they map a subset of those types to OpenCL without the need of type inference. However, they do not support the dynamic Number type. Also, they use ahead-of-time compilation. Ikra [18] implements ahead-of-time compilation of a subset of Ruby to GPUs. Similar to our approach, the authors propose a new data type *PArray* and they also use code analysis to decide whether an operation may run in parallel. However, their system only supports map and inject (reduce) operations and, while showing promising first results, is still in an early stage of development. CopperHead [7] supports ahead-of-time and just-in-time compilation of a subset of Python to CUDA. Their work solves many of the challenges addressed by our solution. However, their compiler is fundamentally a static compiler. In particular, they do not specialize generated code to the extent a dynamic compiler like the one presented here can.

WebCL [1] proposes an embedding of OpenCL into JavaScript at the language rather than implementation level. Unlike River Trail, it is targeted at limited usage scenarios that allow for a violation of JavaScript security model, such as, for example, natively installed HTML5 applications.

Further related work can be found in the context of programming models for deterministic parallelism. In the context of statically typed languages, approaches like DPJ [6] (Java) and CSOLVE [17] (C) have been proposed. Their concurrency model is more general than ours. However, they need program annotations to prove determinism. [19] presents a concurrency model similar to ours that also allows sibling threads read-only access to the parent's state. However, it is tailored towards task parallelism and uses explicit operations to encode sibling/sibling dependencies. The latter cannot arise in our model.

Finally, we have discussed this and related work in earlier publications. [13] is a position paper that argues for the need of a parallel programming model for JavaScript. In [29], we have presented materials that teach concurrency with the help of River Trail. The design process, details of the implementation and results of our in depth performance evaluation have not been published before.

6. Conclusion

We have reported on River Trail, a joint expedition of language designers and web developers towards a parallel programming API for JavaScript. During the design, we have co-evolved the API and its implementation with new workloads and usage scenarios. The resulting API is tailored for the needs of the web: it is safe and secure, builds on existing developer knowledge and offers performance portability. We have described our prototype implementation and outlined the key techniques used to bring River Trail to multi-core CPUs and GPUs. Our evaluation proves that significant performance improvements up to an order of magnitude can be achieved for realistic web applications.

Rapid prototyping of the compiler and runtime during the different design iterations has proven very beneficial, especially to collect early developer feedback. Yet, design should ultimately lead to a finished product. To that effect, we have proposed River Trail to the ECMAScript standards committee to further evolve the API [15]. In parallel, we are working jointly with Mozilla on a production quality implementation.

References

[1] T. Aarnio and M. Bourges-Sèvenier. WebCL working draft. `https://cvs.khronos.org/svn/repos/registry/trunk/public/webcl/spec/latest/index.html`, October 2012.

[2] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari. Transforming the web into a real application platform: new technologies, emerging trends and missing pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 800–807, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0113-8.

[3] S. Avidan and A. Shamir. Seam carving for content-aware image resizing. *ACM Trans. Graph.*, 26, July 2007. ISSN 0730-0301.

[4] R. Berjon, T. Leithead, E. Doyle Navara, E. O'Connor, S. Pfeiffer, and I. Hickson. HTML5. `http://dev.w3.org/html5/spec/`, November 2012.

[5] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, Apr. 1994. ISSN 0743-7315.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0.

[7] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0.

[8] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782.

[10] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 23–23, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4.

[11] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9.

[12] G. Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian conference on Image analysis*, SCIA'03, pages 363–370, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40601-8.

[13] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. Parallel programming for the web. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[14] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. ParallelArray API specification. `https://github.com/`

RiverTrail/RiverTrail/wiki/ParallelArray, October 2012.

[15] R. L. Hudson and S. Herhut. Parallel EcmaScript (River Trail) API. http://wiki.ecmascript.org/doku.php?id=strawman:data_parallelism, October 2012.

[16] K. E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

[17] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala. Deterministic parallelism via liquid effects. *SIGPLAN Not.*, 47(6): 45–54, June 2012. ISSN 0362-1340.

[18] H. Masuhara and Y. Nishiguchi. A data-parallel extension to ruby for gpgpu: toward a framework for implementing domain-specific optimizations. In *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE '12, pages 3–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1277-6.

[19] N. D. Matsakis. Parallel closures: a new twist on an old idea. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, Berkeley, CA, USA, 2012. USENIX Association.

[20] Mozilla. Firefox OS. http://www.mozilla.org/en-US/firefoxos/, November 2012.

[21] A. Munshi. OpenCL specification 1.1. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, June 2011.

[22] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8.

[23] N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for gpus in scala. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 107–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8.

[24] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[25] A. Pignotti, A. Welc, and B. Mathiske. Adaptive data parallelism for internet clients on heterogeneous platforms. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 53–62, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7.

[26] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, Berkeley, CA, USA, 2010. USENIX Association.

[27] 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 '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

[28] S. Singh. Declarative data-parallel programming with the accelerator system. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, DAMP '10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-859-9.

[29] J. Sreeram, S. Herhut, R. L. Hudson, and T. Shpeisman. Teaching parallelism with river trail. In *Proceedings of the 2012 workshop on Developing competency in parallelism: techniques for education and training*, DCP '12, pages 1–8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1840-2.

[30] D. Tiwari and Y. Solihin. Architectural characterization and similarity analysis of sunspider and Google's v8 JavaScript benchmarks. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 221 –232, april 2012.