A SIMD Programming Model for Dart, JavaScript, and other dynamically typed scripting languages

John McCutchan
Google Inc.
johnmccutchan@google.com

Haitao Feng
Intel Corp.
haitao.feng@intel.com

Nicholas D. Matsakis
Mozilla Research
nmatsakis@mozilla.com

Zachary Anderson
Google Inc.
zra@google.com

Peter Jensen
Intel Corp.
peter.jensen@intel.com

Abstract

It has not been possible to take advantage of the SIMD co-processors available in all x86 and most ARM processors shipping today in dynamically typed scripting languages. Web browsers have become a mainstream platform to deliver large and complex applications with feature sets and performance comparable to native applications, programmers must choose between Dart and JavaScript when writing web programs. This paper introduces an explicit SIMD programming model for Dart and JavaScript, we show that it can be compiled to efficient x86/SSE or ARM/Neon code by both Dart and JavaScript virtual machines achieving a 300%-600% speed increase across a variety of benchmarks. The result of this work is that more sophisticated and performant applications can be built to run in web browsers. The ideas introduced in this paper can also be used in other dynamically typed scripting languages to provide a similarly performant interface to SIMD co-processors.

1. Introduction

All x86 and many ARM processors shipping today include a dedicated single instruction multiple data (SIMD) co-processor. On x86 the SSE instruction set allows for computing on 128-bit wide registers. ARM has the Neon instruction set which allows for computing on 128-bit wide registers. Both SSE and Neon have instructions for computing on 4 single precision floating point numbers (`Float32x4`) and 4 signed integers (`Int32x4`) stored in 128-bit registers. There are also instructions available for operating on 2 double precision floating point numbers (`Float64x2`) as well as algorithm specific instructions for operating on sound or pixel data.

Web browsers as a platform to deliver complex applications with feature sets and performance that are comparable to native desktop applications have become mainstream. All major browsers (Chrome, Firefox, Safari, and IE) include support for displaying 3D graphics through WebGL, playing 3D positional audio through WebAudio. These applications are written directly in JavaScript, Dart, or compiled to JavaScript from C/C++ code through the LLVM based Emscripten [4] compiler. Starting in 2008 with the advent of the V8 JavaScript Virtual Machine (VM) and 2009's TraceMonkey [7] JavaScript Virtual machine performance of JavaScript execution has increased dramatically with browser vendors competing for the fastest JavaScript VM. Dart is a new web programming language designed for performance with a custom VM.

Despite the dramatic increase in performance it has not been possible for web programs to access the dedicated SIMD co-processors. JavaScript is limited to scalar operations on double precision floating point values and Dart is limited to scalar operations on double precision floating point values and integers. In this paper we introduce:

- A SIMD programming model for Dart and JavaScript languages. The programming model is high level but allows for explicit control over the SIMD co-processors. It supports comparisons, branchless selection based on comparison results, and efficient shuffling of scalar values within a SIMD value. The programming model does not rely on compiler optimization passes like auto vectorization.
- Three (`Float32x4`, `Int32x4`, and `Float64x2`) new numeric value types for Dart and JavaScript.
- Efficient array storage of the new numeric value types.
- How we generate performant machine code that is free of high level abstraction in Dart and JavaScript VMs.
- An implementation of the programming model for the Dart VM that has shipped and is used in production today.
- An implementation of the programming model for the V8 and SpiderMonkey JavaScript VMs.
- Benchmarks that show (depending on the algorithm) a speedup of 300-600% across problem domains including real-time 3D graphics, numerical computation, ray tracing, and cryptography.

WPMVP'14, February 15–19, 2014, Orlando, FL, USA.

http://dx.doi.org/10.1145/2568058.2568066

```
1 double scalar_average(Float32List data) {
2   var sum = 0.0;
3   for (var i = 0; i < data.length; i++) {
4     sum += data[i];
5   }
6   return sum / data.length;
7 }
```

```
1 double simd_average(Float32x4List data) {
2   var sum = new Float32x4.zero();
3   for (var i = 0; i < data.length; i++) {
4     sum += data[i];
5   }
6   var total = sum.x + sum.y + sum.z + sum.w;
7   return total / (data.length * 4);
8 }
```

Figure 1. Scalar (top) and SIMD (bottom) implementations of an algorithm to find the average of an array of numbers.

2. Lack of SIMD programmability for web applications

Many algorithms can be sped up by taking advantage of SIMD co-processors. A simple example is the averaging of an array of numbers. The top of Figure 1 shows a scalar implementation written in Dart. The algorithm simply accumulates all the data points and computes the average by dividing by the number of data points. The bottom of the figure shows a SIMD implementation also written in Dart. This algorithm can trivially take advantage of SIMD co-processors by adding 4 numbers at the same time.

The bulk of the work is done in parallel and only after exiting the loop does the program need to fall back to scalar computation when computing the final sum and average.

If the `Float32x4` type were available to web programmers and the optimizing compiler is successful in generating code that is free of memory allocation and allows for temporary values to stay in CPU registers, the algorithm can be sped up by 500%. In the following section, we provide more details on the programming model and how it can be efficiently compiled for x86 and ARM processors.

3. Bringing SIMD to the Web

The SIMD programming model for Dart and JavaScript is designed to give direct control to the programmer (or the Dart or C++ compiler generating JavaScript). It introduces three new 128-bit wide value types: `Float32x4`, `Int32x4`, and `Float64x2`. Each value type stores scalar values in multiple "lanes". For example, `Float32x4` has four single precision floating point numbers in lanes labelled: x, y, z, and w. Note that w is the fourth lane and not the first. Each instance is immutable and all operations result in a new instance.

3.1 Primitive Operations

This section will focus on the `Float32x4` type. The other types offer similar operations but can be limited by a lack of actual instructions in SSE or Neon. For example, there is no `Int32x4` divide instruction in either SSE or Neon instruction sets. Emulation of missing instructions is discussed in future work.

`Float32x4` supports standard arithmetic operations (+, -, *, /) as well as approximate square root, reciprocal square root, and reciprocal. It also supports absolute value, minimum, maximum, and clamp operations. All of these operations are performed for each lane. For example, the minimum value of two `Float32x4` is the `Float32x4` with the minimum value of each individual lane.

```
1 num scalar_min(num a, num b) {
2   if (a <= b) {
3     return a;
4   }
5   return b;
6 }
```

```
1 Float32x4 simd_min(Float32x4 a, Float32x4 b) {
2   Int32x4 mask = a.lessThanOrEqual(b);
3   return mask.select(a, b);
4 }
```

Figure 2. The scalar (top) and SIMD (bottom) minimum function.

3.2 Type Conversion

Value cast operations between `Float32x4` and `Int32x4` as well as `Float32x4` and `Float64x2` are available. In the conversion between `Float32x4` and `Float64x2` only the x and y lanes are used.

Bit-wise cast operations between `Float32x4`, `Int32x4`, and `Float64x2` are available. These do not interpret the lane values but provide a mechanism to directly convert the 128-bit value between all type pairs.

3.3 Comparison and Branchless Selection

When comparing SIMD values the result is not a single boolean value but a boolean value for each lane. Consider the example of computing the minimum value of two values. Figure 2 shows the scalar and SIMD algorithms written in Dart:

The comparison results in an `Int32x4` value with lanes containing `0xFFFFFFFF` or `0x0` when the lane comparison is true or false respectively. The resulting mask is used to pick which value's lane should be used.

3.4 Lane Access and Shuffling

Direct access to each lane of a `Float32x4` can be had by accessing the x, y, z, or w instance properties. An example of this is shown in the average algorithm. Shuffling the order of lanes can also be done. Reversing the order of the lanes, in Dart:

```
Float32x4 reverse(Float32x4 v) {
  return v.shuffle(Float32x4.WZYX);
}
```

The shuffle method uses an integer mask to reorder the lane values in v. All 256 combinations are supported.

Because each instance is immutable it is not possible to change the value stored in one of the lanes. Methods, for example, `withX` allow for constructing new instances that are copies of an existing instance with an individual lane value changed. For example, in Dart:

```
var x = new Float32x4(1.0, 2.0, 3.0, 4.0);
var y = x.withX(5.0);
```

3.5 Memory I/O

Up until this point we have only discussed individual values but in order to be a generally useful programming model, compact and cache-friendly array storage of each type is introduced. `Float32x4List`, `Int32x4List`, and `Float64x2List` offer contiguous storage of `Float32x4`, `Int32x4`, and `Float64x2` values. These lists do not store instances but their 128-bit payloads. Figure 3 shows loading and storing SIMD values in Dart.

Note that on the load on line 5 a new instance of `Float32x4` is constructed. In Sections 4.3 and 4.5 we discuss how the memory allocation and instance construction is avoided in optimized code.

```
1 void copy(Float32x4List destination,
2           Float32x4List source,
3           int n) {
4   for (var i = 0; i < n; i++) {
5     var x = source[i]; // Load.
6     destination[i] = x; // Store.
7   }
8 }
```

Figure 3. The `copy` function copies an array of SIMD values.

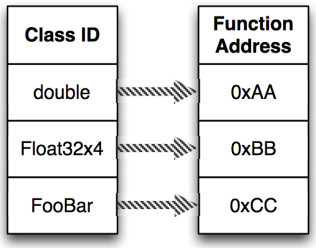

Figure 4. A call-site's type-cache.

3.6 Mapping from high level to low level

The programming model is high level with each operation requiring a method call on a heap allocated object and results in a new heap allocated object holding the resulting value. Each value is immutable and storage of temporary values cannot be reused. The design of the programming model was done with care so that when optimized code is generated, the overhead of the high level programming model can be removed. Almost all method calls will be mapped directly to a single CPU instruction. Some operations require slightly longer instruction sequences but so would a hand written assembly programming performing the same operation. Instances will be stored directly inside CPU registers avoiding the cost of memory allocation and object creation. The following section covers how this is accomplished.

4. Dart VM implementation

We will now discuss the implementation details of the programming model in the Dart VM. Section 4.11 discusses the implementation for JavaScript.

4.1 Unoptimized Code

When a function is first compiled by the Dart VM the generated code is completely generic. Every method call is looked up in the receiving object's class's function table. Every temporary value is allocated in the heap as a full object under control of the GC.

4.2 Type Collection

Using techniques developed by Hölzle et al. [8] and other researchers, the unoptimized code collects important type information that is used later by the optimizing compiler. At each method call the unoptimized code maintains a cache mapping from receiver class id to address of the class's corresponding function, as shown in Figure 4.

This data has two important uses:

- Future execution of the unoptimized code will be faster if the receiving object has a class id already seen at the call site. Execution is faster because looking up the function for the method on the receiving object will result in a cache hit, avoiding the expensive lookup and cache update.

Figure 5. A boxed SIMD value.

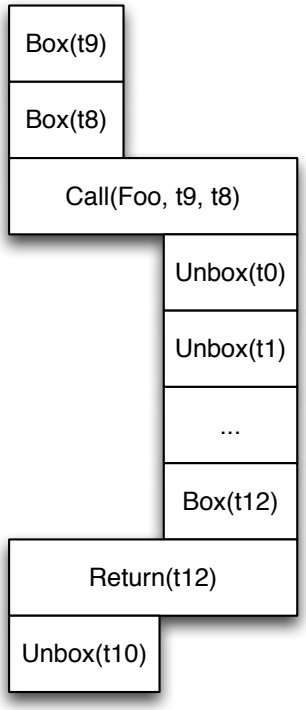

Figure 6. A SIMD value being boxed and unboxed for parameter passing and function return.

- Types seen at a method call site are fed into the optimizing compiler and code that optimistically expects to see the same type can be generated.

4.3 Boxed and Unboxed Values

The Dart compiler makes a distinction between boxed and unboxed [9] values. Boxed values are pointers to objects which are allocated in the heap whose life cycle is managed by the GC. Unboxed values are stored in CPU registers. Operations on unboxed values are much more efficient because the values are already contained in CPU registers. Figure 5 shows the in-memory layout of an instance of `Float32x4`. The object header contains information used for type collection and GC.

4.4 Parameter Passing

The Dart VM passes all method parameters and return value as boxed instances. This can have a negative performance impact as unboxed values will be boxed for the method call which will immediately unbox them again, as shown in Figure 6.

4.5 Inlining

Both the Dart VM and the JavaScript VMs make heavy use of inlining to avoid method call invocation and unnecessary boxing of values. The first form of inlining is similar to the inlining done in many compilers, the bodies of small functions are copied into

```
1 Float32x4 compute(Float32x4 a, Float32x4 b) {
2   var c = a + b;
3   var d = a * a;
4   var e = b * b;
5   return c + d + e;
6 }
```

```
1 t0 ← Call(+, a, b);
2 t1 ← Call(*, a, a);
3 t2 ← Call(*, b, b);
4 t3 ← Call(+, t0, t1);
5 t4 ← Call(+, t3, t2);
6 return t4;
```

Figure 7. A Dart function (top) and its compiled, but unoptimized code (bottom). `Call` here indicates an expensive call into the Dart runtime.

```
1 if (TypeMatch(a, Float32x4)) {
2   deoptimize();
3 }
4 if (TypeMatch(b, Float32x4)) {
5   deoptimize();
6 }
7 t0 ← unbox(a);
8 t1 ← unbox(b);
9 t2 ← BinaryFloat32x4Op+(t0, t1);
10 t3 ← BinaryFloat32x4Op*(t0, t0);
11 t4 ← BinaryFloat32x4Op*(t1, t1);
12 t5 ← BinaryFloat32x4Op+(t2, t3);
13 t6 ← BinaryFloat32x4Op+(t5, t4);
14 return box(t6);
```

Figure 8. The `compute` function after optimizing compilation. Here, `BinaryFloat32x4Op{+,*}` indicate fast, inlined machine code routines.

the calling function, replacing the method call. The second form of inlining, replacement of runtime provided functions with compiler intermediate representation (IR) instructions is the key to achieving high performance with this programming model. Consider the code in Figure 7.

So long as the function `compute` has only ever been given `Float32x4` instances, the compiler will recognize all of the call sites are monomorphic, in other words, each call site has a single receiver class id and target function. The type data collected in unoptimized code at each method call site is used to determine whether this is the case or not. The function targets `Float32x4_Add` and `Float32x4_Mul` are both provided by runtime and the compiler knows how to replace them with IR instructions. As mentioned above, the functions only accept and return boxed instances but the `Float32x4` IR instructions only accept and return unboxed values.

4.6 Optimized Code

When inlining is successful the optimized code will be free of object construction and method call invocations. Figure 8 shows the `compute` function from Figure 7 after optimization.

The optimized code first validates the class of each input value. If any of the types do not match the expected type, the function is deoptimized. For more details on deoptimization see Section 4.9. After being validated the values are unboxed directly into CPU registers (tN) and the remaining operations are performed directly in CPU registers with no memory I/O. The final step is to box the result so it can be returned. Figure 9 shows the final assembled result using Intel SSE SIMD instructions.

For readers unfamiliar with SSE, the following list describes the instructions used in the example:

```
        ;; CheckClass:14(v2)
0x107822533 testb rdx,0x0x1
0x107822536 jz 0x107822607
0x10782253c movzxwq rbx,[rdx+0x1]
0x107822541 cmpl rbx,0x37
0x107822544 jnz 0x107822607
        ;; CheckClass:14(v3)
0x10782254a testb rcx,0x0x1
0x10782254d jz 0x107822612
0x107822553 movzxwq rbx,[rcx+0x1]
0x107822558 cmpl rbx,0x37
0x10782255b jnz 0x107822612
        ;; v11 <- UnboxFloat32x4:14(v2)
0x107822561 movups xmm1, [rdx+0x7]
        ;; v14 <- UnboxFloat32x4:14(v3)
0x107822565 movups xmm2, [rcx+0x7]
        ;; ParallelMove xmm3 <- xmm1
0x107822569 movaps xmm3, xmm1
        ;; v4 <- BinaryFloat32x4Op:14(+, v11, v14)
0x10782256c addps xmm3, xmm2
        ;; ParallelMove xmm4 <- xmm1
0x10782256f movaps xmm4, xmm1
        ;; v5 <- BinaryFloat32x4Op:26(*, v11, v11)
0x107822572 mulps xmm4, xmm1
        ;; ParallelMove xmm1 <- xmm2
0x107822575 movaps xmm1, xmm2
        ;; v6 <- BinaryFloat32x4Op:38(*, v14, v14)
0x107822578 mulps xmm1, xmm2
        ;; v7 <- BinaryFloat32x4Op:50(+, v4, v5)
0x10782257b addps xmm3, xmm4
        ;; v8 <- BinaryFloat32x4Op:58(+, v7, v6)
0x10782257e addps xmm3, xmm1
        ;; v15 <- BoxFloat32x4:112(v8)
0x107822581 movq r11,[r15+0x47]
0x107822588 movq rax,[r11]
0x10782258b addq rax,0x20
0x10782258f movq r11,[r15+0x4f]
0x107822596 cmpq rax,[r11]
0x107822599 jnc 0x1078225eb
0x10782259f movq r11,[r15+0x47]
0x1078225a6 movq [r11],rax
0x1078225a9 subq rax,0x1f
0x1078225ad movq [rax-0x1],0x370200
0x1078225b5 movups [rax+0x7], xmm3
```

Figure 9. Optimized machine code generated by the Dart VM for the `compute` function, SSE instructions bolded.

- **movups** Unaligned load or store of 128-bit value.
- **movaps** Register move.
- **addps** Add 4 single precision floating point numbers.
- **mulps** Multiply 4 single precision floating point numbers.

4.7 Register Allocation

The Dart VM already supported unboxed double values which on x86 used the xmm registers. The xmm registers are also needed for `Float32x4`, `Int32x4`, and `Float64x2` values. The register allocator was extended to support both 8 and 16 byte values stored in `xmmN` registers as well as allocate the appropriate range of memory to spill values from registers to the stack when register pressure requires spilling.

4.8 Alignment

SIMD instruction sets provide preferred memory I/O instructions that require the memory address to be aligned on 16-byte boundaries. These instructions are typically faster than the unaligned memory I/O instructions. All memory I/O instructions emitted by

```
1 function compute(a, b) {
2   var c = SIMD.float32x4.add(a, b);
3   var d = SIMD.float32x4.mul(a, a);
4   var e = SIMD.float32x4.mul(b, b);
5   var t = SIMD.float32x4.add(c, d);
6   return SIMD.float32x4.add(t, e);
7 }
```

Figure 10. Our `compute` function translated to JavaScript using our experimental SIMD module.

the Dart VM are for unaligned addresses because the Dart VM GC cannot guarantee object alignment.

4.9 Deoptimization

After a function has been optimized if one of the values it uses violates the optimistic type expectations, a deoptimization occurs resulting in the following process:

- Execution of optimized code stops.
- Optimized code is invalidated and disconnected from the function.
- Live unboxed values are boxed.
- Pointers to boxed values are stored in the correct locations of the unoptimized codes stack frame.
- Execution of the function's unoptimized code begins at exactly the same point that the deoptimization occurred.

An important thing to understand is that only functions which are provided stable input types will be optimized.

4.10 Port to ARM/Neon

We have ported these SIMD operations in Dart to the Neon instruction set provided by many ARM implementations. In most cases the mapping from SSE instructions to Neon instructions is straightforward. In the case of the shuffle operation, the Neon instruction set lacks a fully general single instruction. In this, and a limited number of additional cases, we take advantage of the overlap of the 128-bit "Q" registers with the 64-bit "D" and 32-bit "F" registers, and implement shuffles, etc., by simply manipulating the "D" and "F" registers that underly the "Q" registers. Presently, these sequences are generated naively, and we are unaware of the performance cost, though we assume it is small based on the benchmark results in Section 5. We leave as future work finding minimal Neon sequences to implement the more general instructions provided by SSE.

4.11 Implementation for JavaScript

The SIMD programming model for Dart is used for JavaScript as well. The 128-bit wide value types are introduced, but as there are no class and operator overloading language features in JavaScript, the syntax of the SIMD API is different than Dart's. We introduced a SIMD global module with operations grouped by type. The following example demonstrates the syntactic differences. Figure 10 shows the translation of our `compute` function into JavaScript.

4.11.1 Implementation in V8

The V8 implementation is conceptually similar to the implementation in the Dart VM:

- We added the 128-bit wide value constructor and SIMD API into full-codegen which generates un-optimized code and optimized them in the Crankshaft which generates optimized code.
- We allocated the 128-bit value in a heap object (with a header to indicate the type) and garbage collect it when it is boxed; when the value is un-boxed, we allocate it into a hardware 128-bit register (XMM register in the Intel IA32 and X64 architectures).
- We inlined all the SIMD operations, inlining is the key technique for the performance acceleration.
- Like the Dart VM, when transitioning from optimized to unoptimized code (due to a deoptimization event) the unboxed values are boxed and pointers to the boxed values are placed in the unoptimized codes stack frame.

Despite the many similarities, there are some important differences in the V8 implementation:

- The unoptimized code does not collect type information for `Float32x4`, `Int32x4`, and `Float32x4` instances. Instead upon entry to a SIMD.type.op function the types of the parameters are verified to match the expected type. If they do not an exception is thrown.
- Because JavaScript is more dynamic than Dart and allows for object fields and functions to be modified at runtime, for example, the `SIMD.float32x4.add` method could have been overridden by the program. More checks are required to ensure that these operations have not been replaced with user written code. When this occurs inlining cannot be performed and performance suffers.

4.11.2 Implementation in SpiderMonkey

We are currently in the process of implementing the SIMD model for the SpiderMonkey engine – used in the Firefox web browser – as well. At the time of this writing, the SpiderMonkey implementation is able to optimize several benchmarks, but it is not yet as mature as the V8 or Dart implementations.

- In the interpreter, 128-bit wide values are represented as objects, each of which contains a pointer to a memory block storing the SIMD value. This is not entirely faithful to the proposed specification, in which the SIMD values are not objects but rather a new kind of primitive JS value. Work on extending the set of primitive JS types that SpiderMonkey supports is already underway, and we plan to leverage that work once it completes so as to adhere more closely to the specification.
- In JIT-optimized code, the SIMD values are aggressively unboxed. Calls to SIMD operations are detected and replaced with native variants that operate directly on the unboxed SIMD values. Extensions were needed to the register allocator and bailout mechanism so that they could support 128-bit values.
- If a bailout occurs, the unboxed SIMD values must be boxed into objects so that the interpeter can resume execution. The same is true whenever an unboxed value escapes out of the current function being compiled, with the exception of stores into SIMD arrays.

5. Benchmarks

In this section we introduce a variety of application domains that can benefit from SIMD arithmetic. We discuss a variety of benchmarks that we have implemented. Finally, we present benchmark results.

5.1 Application Domains

At this stage in our implementation, benchmarks should have the following two goals:

1. They should reflect meaningful operations typically done with SIMD programming,

2. They should be small and easy to use for VM implementers to drive the performance work in the VMs, when implementing JIT compiler support for those operations.

For the benchmarks to be meaningful, they should cover at least some of the base functionality required for application domains that benefit from SIMD programming. Some (not all) of those domains are; 2D graphics (e.g. filters, rendering) and 3D graphics (e.g. WebGL). There are other domains, such as computational fluid dynamics, finance (Black-Sholes), cryptography, video processing, audio processing, etc. However, we have not yet extracted meaningful kernels for typical SIMD kernels from all of those domains. For the benchmarks to be small, they should mostly target select individual operations. The current set of benchmarks cover the following application domains:

- **3D Graphics** This application domain is tailor made for use of Float32x4 operations, because most of the arithmetic involved operates on 4x4 or 3x3 matrices and vectors. We've picked a few of the common operations to showcase the impact of SIMD. The operations covered so far are:
  - **Vertex Transformation** Transformation of 4 element vector by 4x4 matrix,
  - **Matrix Transpose** Transposition of 4x4 matrix,
  - **Matrix Inversion** Compute the inverse of a 4x4 matrix,
  - **Matrix Multiplication** Multiply two 4x4 matrices.
- **Cryptography** This might not be a typical domain where use of SIMD operations can improve performance, but we found a hot function in the implementation of the Rijndael cipher [6] that would benefit, so we extracted it into a benchmark kernel.
  - **Shift Rows** (shift/rotate of row values in 4x4 matrix)
- **Vector Math Computations** Math operations on small vectors, beyond simple algebraic operations (+,-,*,/), typically include fairly complicated uses of SIMD operations, i.e. not just individual SIMD instructions. Intel provides a Short Vector Math Library [1] for that purpose for use in C/C++ SIMD code. It is envisioned that such base math operations (sin, cos, exp, pow, etc) will be useful for things like signal processing and physics engines. In order to show that a similar library could be implemented using the SIMD programming API in Dart and JavaScript, with performance improvements equivalent to that of highly optimized C/C++ functions, we have implemented a benchmark that computes sine.
  - **Sinex4** Compute 4 sine values at once.

5.2 Results

In this section we present benchmark results for our benchmarks. Full source code of these benchmarks is available from a GitHub repository [3]. In addition to the benchmarks discussed above, we also include a simple average microbenchmark as well as a larger Mandelbrot [2] visualization.

We ran these benchmarks on the Dart VM, V8, and SpiderMonkey using Dart and JavaScript implementations as appropriate. We used an Ubuntu 13.04 64-bit Linux system with an Intel CPU (Intel(R) Xeon(R) CPU E5-2690 @ 2.90GHz). We also ran the benchmarks on the Dart VM running on a Nexus 7, ARM Android device. Since ports to ARM/Neon are not yet complete for V8 and SpiderMonkey we did not run the JavaScript implementations on the Nexus 7.

We ran each benchmark repeatedly until at least 2 seconds had elapsed, then divided by the number of iterations in this time to arrive at the time taken for a single iteration. We report absolute times in microseconds for both the scalar and SIMD versions of the benchmarks, as well as the SIMD speedup calculated as the scalar time divided by the SIMD time.

Benchmark	Scalar Time(us)	SIMD Time(us)	Speedup
Dart			
Average	381	36	**10.5x**
Mandelbrot	330714	128750	**2.6x**
MatrixMultiply	68	17	**4.0x**
MatrixInverse	181	29	**6.2x**
MatrixTranspose	1220	523	**2.3x**
VectorTransform	25	5	**5.0x**
ShiftRows	5658	623	**9.1x**
V8			
Average	208	35	**5.9x**
Mandelbrot	393167	109158	**3.6x**
MatrixMultiply	74	20	**3.7x**
MatrixInverse	189	21	**9.0x**
MatrixTranspose	1037	408	**2.5x**
VectorTransform	30	6	**5.0x**
ShiftRows	6067	880	**6.9x**
AOBench	1488	736	**2.0x**
SineX4	9538	6568	**1.5x**
SpiderMonkey			
Average	116	21	**5.5x**
Mandelbrot	346333	152357	**2.3x**
MatrixMultiply	97	19	**5.1x**
MatrixInverse	294	26	**11.3x**
MatrixTranspose	1237	488	**2.5**
VectorTransform	33	8	**4.1x**
ShiftRows	6067	1956	**3.1x**

Table 1. Benchmark results for Dart, V8, and SpiderMonkey on our x64 Linux machine.

Benchmark	Scalar Time(us)	SIMD Time(us)	Speedup
Average	1832	180	**10.1x**
Mandelbrot	1806000	892333	**2.0x**
MatrixMultiply	630	224	**2.8x**
MatrixInverse	1506	345	**4.4x**
MatrixTranspose	6335	5488	**1.2x**
VectorTransform	173	67	**2.6x**
ShiftRows	33148	3219	**10.3x**

Table 2. Benchmark results for Dart on our ARM Android device.

5.3 Discussion

The results of our benchmarks are shown in Tables 1 and 2. Table 1 shows results on our Intel machine for Dart, V8, and SpiderMonkey. Table 2 shows results on a Nexus 7 device for Dart [1]. The benchmark data show significant speedups across a variety of algorithms, implementations and across the two platforms when we use SIMD operations. Further, our JavaScript and Dart benchmark kernels and timing harness are as similar as possible, so it is possible not only to compare scalar and SIMD implementations running on the same VM, but also to compare results across the three VMs.

[1] ARM ports of SIMD for V8 and SpiderMonkey are not yet complete.

Thus, some of the results require some additional explanation. The theoretical performance gain from SIMD should only be 4x, however a number of our benchmarks show larger gains. It is important to note that in the case of Dart and JavaScript that the scalar code performs double precision arithmetic, while the SIMD code performs single precision arithmetic. The data in these scalar benchmarks is loaded from an array holding single precision floating point values. Thus, the scalar code must convert from single to double precision before performing the calculation, while the SIMD code requires no conversion, implying some saved time for the SIMD codes.

As for the Mandelbrot benchmark, Dart and SpiderMonkey achieved only speedups of 2.6x and 2.3x respectively, while V8 achieved a speedup of 3.6x. The reason for the discrepancy is that the Dart VM has not fully implemented the necessary inlining optimizations for the `Int32x4` type which is used in the Mandelbrot benchmark. At this early stage, we find these benchmark results encouraging, and that they will be useful in guiding our further implementation efforts.

6. Related Work

6.1 C SSE Intrinsics

C/C++ compilers offer SIMD intrinsic functions (`xmmintrin.h` and `emmintrin.h`). At compile time these function calls are replaced with a single instruction. Conceptually, the SIMD programming model for Dart and JavaScript is similar but neither Dart nor JavaScript has static type information available at compile time and instead, must rely on run-time type collection (or enforcement) before the compiler can emit the SSE instructions.

6.2 SIMD in C#

The Mono C# compiler provided an API for SIMD [5] programming. This API is considered experimental and directly mimics the SSE instruction set without any concern for portability across different SIMD instruction sets. The SIMD programming model presented in this paper was designed to run across SSE and Neon instruction sets and has been implemented for both.

6.3 Auto-Vectorization

Mozilla has experimented [10] with an auto vectorization pass in their JavaScript compiler. The focus of this work was on a gaussian blur algorithm written in JavaScript. Speed gains of approximately 20% were achieved. This work was never merged and appears abandoned.

7. Conclusions and Future Work

In this paper we have presented a high level programming model that offers direct control of SIMD co-processors to web programs in both Dart and JavaScript. We have shown how both Dart and JavaScript VMs can avoid memory allocation and the resulting garbage collector (GC) pressure by avoiding instance creation with inlining and unboxed values. We have discussed the implementations in both the Dart VM and V8. Finally, we have presented benchmarks demonstrating the real world performance gains that this programming model provides. Future work includes:

- Support for wider (256-bit and 512-bit) SIMD register widths (AVX and AVX-512) and the natural data types they offer, for example, `Float32x8` and `Float32x16`. This includes generating correct code for wider register widths on CPUs that only support 128-bit wide registers.
- Support for pixel processing by developing a portable abstraction around instructions that are algorithm specific operating on 16 individual bytes. Including support for saturating arithmetic operations.
- Support operations which do not have CPU instructions available, for example, Int32x4 division.
- Supporting for passing and returning unboxed values directly, avoiding the overhead of boxing values on method invocation and return.
- Support types that have few SIMD instructions, for example, `Int16x8`. Speed ups may be seen on these types because efficient native code can that does N scalar operations can be emitted directly.

8. Acknowledments

We would like to thank Mohammad R Haghighat for the technical direction, Heidi Pan for the SIMD benchmark (aobench.js), Ivan Jibaja for contributing the Firefox IonMonkey implementation, Ningxin Hu and Weiliang Lin for contributing the Google V8 implementation.

References

[1] Implement the short vector math library, November 2010. http://software.intel.com/en-us/articles/implement-the-short-vector-math-library.

[2] Benoit Mandelbrot, December 2013. http://en.wikipedia.org/wiki/Benoit_Mandelbrot.

[3] ECMAScript SIMD polyfill, January 2014. https://github.com/johnmccutchan/ecmascript_simd/.

[4] Emscripten, January 2014. http://en.wikipedia.org/wiki/Emscripten.

[5] Mono Documentation: Mono.Simd Namespace, January 2014. http://docs.go-mono.com/?link=N%3aMono.Simd.

[6] Joan Daemen and Vincent Rijmen. The design of Rijndael:AES–the advanced encryption standard. *Journal of Cryptology*, 4(1):3–72, 1991.

[7] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[8] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996.

[9] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 177–188, New York, NY, USA, 1992. ACM.

[10] Nicolas B. Pierron. IonMonkey: Use SIMD to optimize gaussian blur, January 2014. https://bugzilla.mozilla.org/show_bug.cgi?id=832718.