Bringing **SIMD-128** to JavaScript

John McCutchan (Google)
Peter Jensen (Intel)
Niko Matsakis (Mozilla)
What is **SIMD**?

... and why does it **matter**?
What is **SIMD**?

Single Instruction Single Data (SISD)
What is **SIMD**?

**Single Instruction Multiple Data (SIMD)**

![Diagram showing SIMD operation]

Vector Processor
Why does **SIMD** matter?

- **SIMD** can provide substantial speedup to:
  - 3D Graphics
  - 3D Physics
  - Image Processing
  - Signal Processing
  - Numerical Processing
  - Crypto
  - Computer Vision
  - ...
Why does **SIMD** matter to the web?

- **SIMD** can provide substantial speedup to:
  - WebGL
  - Canvas
  - Animation
  - Games
  - Physics
  - ASM.js
  - Crypto
Why does **SIMD** matter?
Why does SIMD matter?

- SIMD requires fewer instructions to be executed
  - Fewer instructions means longer battery life
// z(i+1) = z(i)^2 + c
// terminate when |z| > 2.0
// returns 4 iteration counts

function mandelx4(c_re4, c_im4) {
  var z_re4 = c_re4;
  var z_im4 = c_im4;
  var four4 = SIMD.float32x4.splat(4.0);
  var two4 = SIMD.float32x4.splat(2.0);
  var count4 = SIMD.int32x4.splat(0);
  var one4 = SIMD.int32x4.splat(1);

  for (var i = 0; i < max_iterations; ++i) {
    var z_re24 = SIMD.float32x4.mul(z_re4, z_re4);
    var z_im24 = SIMD.float32x4.mul(z_im4, z_im4);
    var mi4 = SIMD.float32x4.lessThanOrEqual
      (SIMD.float32x4.add(z_re24, z_im24), four4);
    // if all 4 values are greater than 4.0
    // there's no reason to continue
    if (mi4.signMask === 0x00) {
      break;
    }
    var new_re4 = SIMD.float32x4.sub(z_re24, z_im24);
    var new_im4 = SIMD.float32x4.mul
      (SIMD.float32x4.mul(two4, z_re4), z_im4);
    z_re4 = SIMD.float32x4.add(c_re4, new_re4);
    z_im4 = SIMD.float32x4.add(c_im4, new_im4);
    count4 = SIMD.int32x4.add(count4, SIMD.int32x4.and (mi4, one4));
  }
  return count4;
}
SIMD-128 for EcmaScript
**SIMD in JavaScript**

- Based on work for Dart Language
  - Landed in Dart VM in Spring of 2013

- Fixed 128-bit vector types as close to the metal while remaining portable
  - SSE
  - NEON
  - Efficient scalar fallback could be implemented

- Scales with other forms of parallelism (e.g. Web Workers)

- Polyfill + benchmarks
  - [https://github.com/johnmccutchan/ecmascript_simd](https://github.com/johnmccutchan/ecmascript_simd)
**SIMD in JavaScript**

1. SIMD module  
   a. New “value” types  
   b. Composable operations  
      i. Arithmetic  
      ii. Logical  
      iii. Comparisons  
      iv. Reordering (shuffling)  
      v. Conversions  
2. Extension to Typed Data  
   a. A new array type for each  

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float32x4</td>
<td>4 IEEE-754 32-bit Floating Point Numbers</td>
</tr>
<tr>
<td>int32x4</td>
<td>4 32-bit Signed Integers</td>
</tr>
<tr>
<td>float64x2</td>
<td>2 IEEE-754 64-bit Floating Point Numbers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float32x4Array</td>
<td>Typed Array of float32x4</td>
</tr>
<tr>
<td>Int32x4Array</td>
<td>Typed Array of int32x4</td>
</tr>
<tr>
<td>Float64x2Array</td>
<td>Typed Array of float64x2</td>
</tr>
</tbody>
</table>
Object Hierarchy

- SIMD
  - int32x4
    - add()
    - sub()
    - ...
  - float32x4
    - add()
    - sub()
    - ...
  - float64x2
    - add()
    - sub()
    - ...
SIMD-128 for EcmaScript Code Snippets
**SIMD in JavaScript**

- `float32x4` (128-bits)
- "lanes"
Constructing

```plaintext
var a = SIMD.float32x4(1.0, 2.0, 3.0, 4.0);

```

```plaintext
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>
```

```plaintext
var b = SIMD.float32x4.zero();

```

```plaintext
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
```
Accessing and Modifying Individual Elements

```
var a = SIMD.float32x4(1.0, 2.0, 3.0, 4.0);

var b = a.x; // 1.0

var c = a.withX(5.0);
```
var a = SIMD.float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.float32x4(5.0, 10.0, 15.0, 20.0);
var c = SIMD.float32x4.add(a, b);
Example

function average(list) {
  var n = list.length;
  var sum = 0.0;
  for (int i = 0; i < n; i++) {
    sum += list[i];
  }
  return sum / n;
}

function average(f32x4list) {
  var n = f32x4list.length;
  var sum = SIMD.float32x4.zero();
  for (int i = 0; i < n; i++) {
    sum = SIMD.float32x4.add(sum, f32x4list.getAt(i));
  }
  var total = sum.x + sum.y + sum.z + sum.w;
  return total / (n * 4);
}
Example
SIMD in JavaScript

75% fewer loads
75% fewer adds
(+ single precision)

5 times faster!
The inner loop

```c
sum = SIMD.float32x4.add(sum, float32x4list.getAt(i));
```
Shuffling

```javascript
var a = SIMD.float32x4(1.0, 2.0, 3.0, 4.0);

var b = SIMD.float32x4.shuffle(a, SIMD.float32x4.XXYY);

var c = SIMD.float32x4.shuffle(a, SIMD.float32x4.WWWW);

var d = SIMD.float32x4.shuffle(a, SIMD.float32x4.WZYX);
```
Branching

```javascript
max = function(a, b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

max(4.0, 5.0) -> 5.0
```
Branching

max = function(a, b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

1.0 2.0 3.0 4.0
0.0 3.0 5.0 2.0
max = function(a, b) {
  var greaterThan = SIMD.float32x4.greaterThan(a, b);
  return SIMD.float32x4.select(a, b, greaterThan);
}
max = function(a, b) {
    var greaterThan = SIMD.float32x4.greaterThan(a, b);
    return SIMD.float32x4.select(a, b, greaterThan);
}
Implementations
How does the VM optimize for **SIMD**?

1. **Unboxing**
   a. Boxed -> allocated in memory
   b. Unboxed -> in CPU memory (in registers)

2. **Replacing method calls with inlined machine instructions**
   a. Allows values to remain unboxed (in registers)
   b. Avoids method call overhead
Firefox implementation status

- **Interpreter support:**
  - In Nightly since early 2014. No flags needed

- **IonMonkey:**
  - Support has been prototyped for x86
  - Missing ARM port of register allocator
  - Ongoing refactoring of a generic register allocator before landing the JIT compiler support
  - Reuse work done for OdinMonkey

- **OdinMonkey (for asm.js):**
  - Current focus
  - Full x86 support planned for end of August in Nightly
Chrome/V8 implementation status

- Code is also hosted in Crosswalk Runtime fork:
  - [https://github.com/crosswalk-project/v8-crosswalk](https://github.com/crosswalk-project/v8-crosswalk)
- Full implementation for polyfill spec:
  - Optimized implementation for ia32 and x64 (full-codegen and crankshaft)
  - Runtime support for ARM/NEON (full-codegen)
- Patches available for:
  - Chrome 34, 35, 36
  - Rebasing for Chrome 37 in progress
Emscripten implementation status

- Supports both the JS and fastcomp ‘backends’
- Supports SIMD.float32x4 and SIMD.int32x4 operations for LLVM vector types:
  - <4 x i32> and <4 x float> LLVM vector types supported
  - Code generated by Loop Vectorizer and SLP Vectorizer
  - Code generated from use of ext_vector_type and vector_size attributes
- Supports a few C++ intrinsics:
  - Most of _mm_<op>_ps (_mm_add_ps, _mm_sub_ps, …)
  - Most of _mm_<op>_epi32 (_mm_add_epi32, _mm_sub_epi32, …)
- No support for x4 arrays, yet
Intel Crosswalk implementation status

- Crosswalk 5,6,7: Full support for polyfill spec
- Crosswalk 8: In progress
- Available via Intel’s XDK build feature
- Optimized for ia32 and x64
- Functional for ARM/NEON
V8 SSE Benchmarks (Early 2014)
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scalar Time (us)</th>
<th>SIMD Time (us)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>208</td>
<td>35</td>
<td>5.9</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>393167</td>
<td>109158</td>
<td>3.6</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>74</td>
<td>20</td>
<td>3.7</td>
</tr>
<tr>
<td>MatrixInverse</td>
<td>189</td>
<td>21</td>
<td>9.0</td>
</tr>
<tr>
<td>MatrixTranspose</td>
<td>1037</td>
<td>408</td>
<td>2.5</td>
</tr>
<tr>
<td>VectorTransform</td>
<td>30</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>ShiftRows</td>
<td>6067</td>
<td>880</td>
<td>6.9</td>
</tr>
<tr>
<td>AOBench</td>
<td>1488</td>
<td>736</td>
<td>2.0</td>
</tr>
<tr>
<td>SineX4</td>
<td>9538</td>
<td>6568</td>
<td>1.5</td>
</tr>
</tbody>
</table>
SpiderMonkey SSE Benchmarks (Early 2014)
### SpiderMonkey SSE Benchmarks (Early 2014)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scalar Time (us)</th>
<th>SIMD Time (us)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>116</td>
<td>21</td>
<td>5.5</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>346333</td>
<td>152357</td>
<td>2.3</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>97</td>
<td>19</td>
<td>5.1</td>
</tr>
<tr>
<td>MatrixInverse</td>
<td>294</td>
<td>26</td>
<td>11.3</td>
</tr>
<tr>
<td>MatrixTranspose</td>
<td>1237</td>
<td>488</td>
<td>2.5</td>
</tr>
<tr>
<td>VectorTransform</td>
<td>33</td>
<td>8</td>
<td>4.1</td>
</tr>
<tr>
<td>ShiftRows</td>
<td>6067</td>
<td>1956</td>
<td>3.1</td>
</tr>
</tbody>
</table>
Dart VM* NEON Benchmarks (Early 2014)
## Dart VM* NEON Benchmarks (Early 2014)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scalar Time (us)</th>
<th>SIMD Time (us)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>1832</td>
<td>180</td>
<td>10.1</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1806</td>
<td>892333</td>
<td>2.0</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>630</td>
<td>224</td>
<td>2.8</td>
</tr>
<tr>
<td>MatrixInverse</td>
<td>1506</td>
<td>345</td>
<td>4.4</td>
</tr>
<tr>
<td>MatrixTranspose</td>
<td>6335</td>
<td>5488</td>
<td>1.2</td>
</tr>
<tr>
<td>VectorTransform</td>
<td>175</td>
<td>67</td>
<td>2.6</td>
</tr>
<tr>
<td>ShiftRows</td>
<td>33148</td>
<td>3219</td>
<td>10.3</td>
</tr>
</tbody>
</table>
Why fixed width and not variable width vectors?

- **Practicality**
  - Stream processing and auto vectorization have limited use cases
  - Variable width vectors cannot efficiently implement
    - Matrix multiplication
    - Matrix inversion
    - Vector transform
    - ....

- **Portable performance**
  - 128-bit is the only vector width supported by all architectures
Why fixed width and not variable width vectors (continued)?

- C/C++ code is usually written using intel _mm_intrinsics.
  - JavaScript as a C/C++ compilation target needs fixed width vectors

- Abstraction
  - Stream processors can be built in software on top of SIMD-128

- Observable state will be different for different architectures, e.g., if SIMD.float32xN was introduced, this code:

```javascript
for (var i = 0; i < M; i += SIMD.float32xN.size) {
    sum = SIMD.float32xN.add(sum, input[i]);
}
```

would cause bits in memory to be different for different architectures.
**SIMD in JavaScript** Miscellaneous

- Result of `typeof`:
  - "float32x4", "float64x2", "int32x4"

- Result of `SIMD.float32x4(1,2,3,4).toString()`:
  - "float32x4(1,2,3,4)"

- Implicit type conversions kept to a minimum:
  - `1 + <float32x4>`:
    - Apply `.toString()` to `<float32x4>` and concatenate
  - `SIMD.float32x4.add(1,<float32x4>)`
    - `TypeError`
**SIMD in JavaScript Planned Features**

- SIMD and value objects/types:
  - float32x4 and friends will be value objects
  - Overloaded operators (+,-,...) will be mapped to SIMD.<type>.<op> equivalents (.add(), .sub(), ...)

- Additional data types (int8x16 and int16x8)
  - Looking at VP9 encode/decode for justification
SIMD in JavaScript Planned Features

- **Feature detection:**
  - Fine grained feature detection
    - Something like: SIMD.optimized.<feature>
  - There are arch differences that will need exposure!
    - Two vector shuffle (Useful for 4x4 matrix transpose)
    - .signmask for NEON
    - Algorithm specific instructions where no overlap/equivalent exists
  - Inlined scalar fallbacks can help minimize performance hit across ISAs
Stage 1 Ready?

✓ Identified “champion” who will advance the addition
✓ Prose outlining the problem or need and the general shape of a solution
✓ Illustrative examples of usage
✓ High-level API
✓ Discussion of key algorithms, abstractions and semantics
✓ Identification of potential “cross-cutting” concerns and implementation challenges/complexity
Wrap up
Concrete implementation, accelerating real world algorithms.

- **SSE***
  - V8
  - SpiderMonkey
  - Intel’s Crosswalk HTML5 runtime
- **NEON**
  - SpiderMonkey (In progress)
  - Dart VM*
Future Work

- SIMD-256 and SIMD-512 extensions
  - No NEON support
    - ARM64 did not extend vector width
    - Can lower SIMD-256 and SIMD-512 operations on to SIMD-128
  - Relevant for server side
  - Lower priority
Questions!
Polyfill repository
https://github.com/johnmccutchan/ecmascript_simd

Published Paper on Dart + JS prototype implementations

HTML5 Developer Conference Presentation (May 2014)
http://peterjensen.github.io/html5-simd/html5-simd.html#

Wikipedia
http://en.wikipedia.org/wiki/SIMD