intel Look Inside™

SIMD in JavaScript via C++ and Emscripten

February 8th, 2015 - Workshop on Programming Models for SIMD/Vector Processing

Peter Jensen, Intel Corporation
Ivan Jibaja, Intel Corporation
Ningxin Hu, Intel Corporation
Dan Gohman, Mozilla
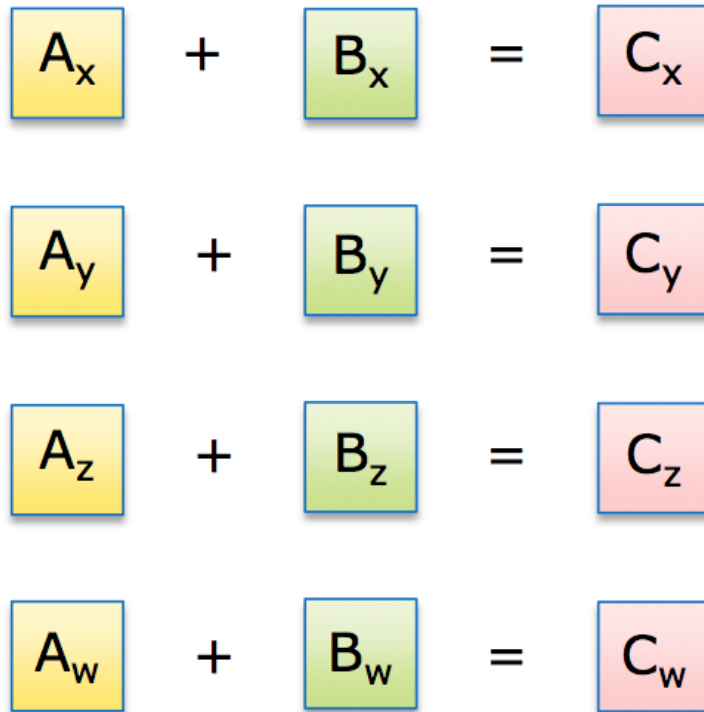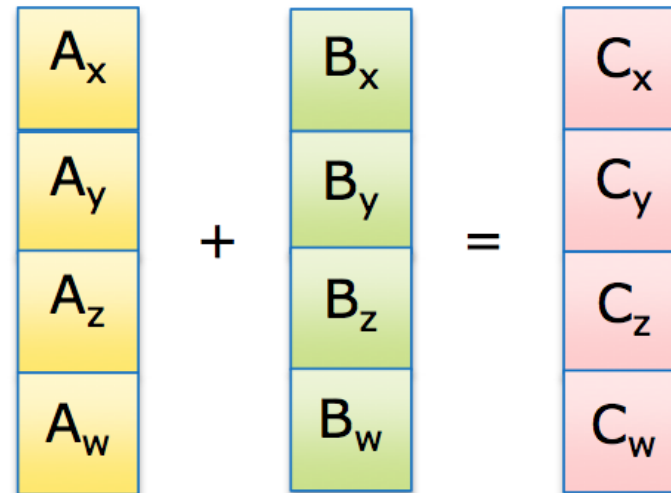**John McCutchan, Google**

Agenda

- Motivation

- Background/History

- SIMD.JS

- Emscripten

- Compiling SIMD C++ code to SIMD.JS JavaScript code

- Benchmark Results

- Summary

SIMD: Single Instruction, Multiple Data

Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

JavaScript's Popularity and Use on the Rise!

- Games (Unreal, Unity) (via Emscripten/asm.js)

- Hybrid HTML5/JS apps for cross platform apps on mobile devices

- Pure HTML5/JS apps on ChromeOS/FirefoxOS/Tizen

- Standalone desktop JavaScript apps via NW.js (formerly node-webkit) (Intel XDK)

- Full featured browser based apps (Google Docs/maps, Office 365, …)
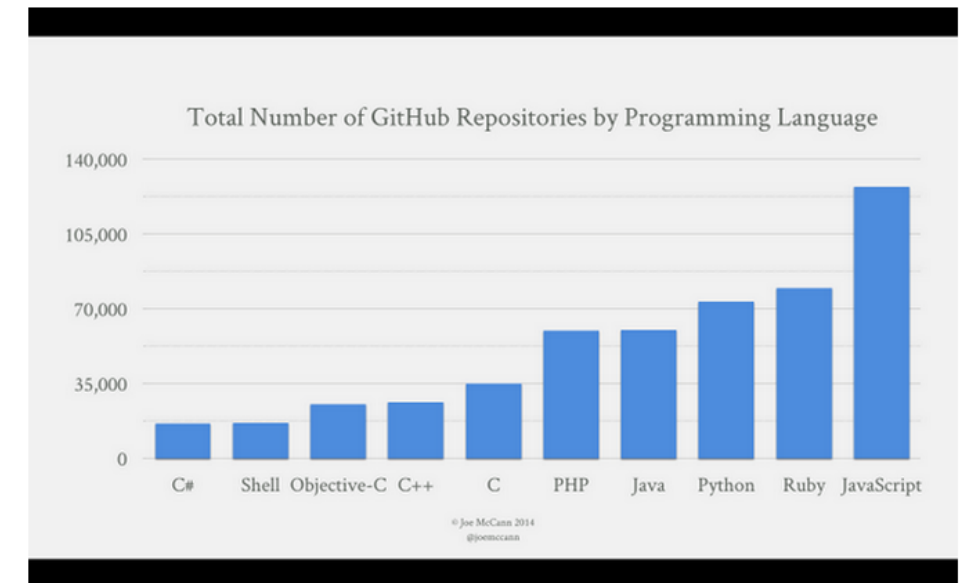
- Server side logic via node.js/io.js
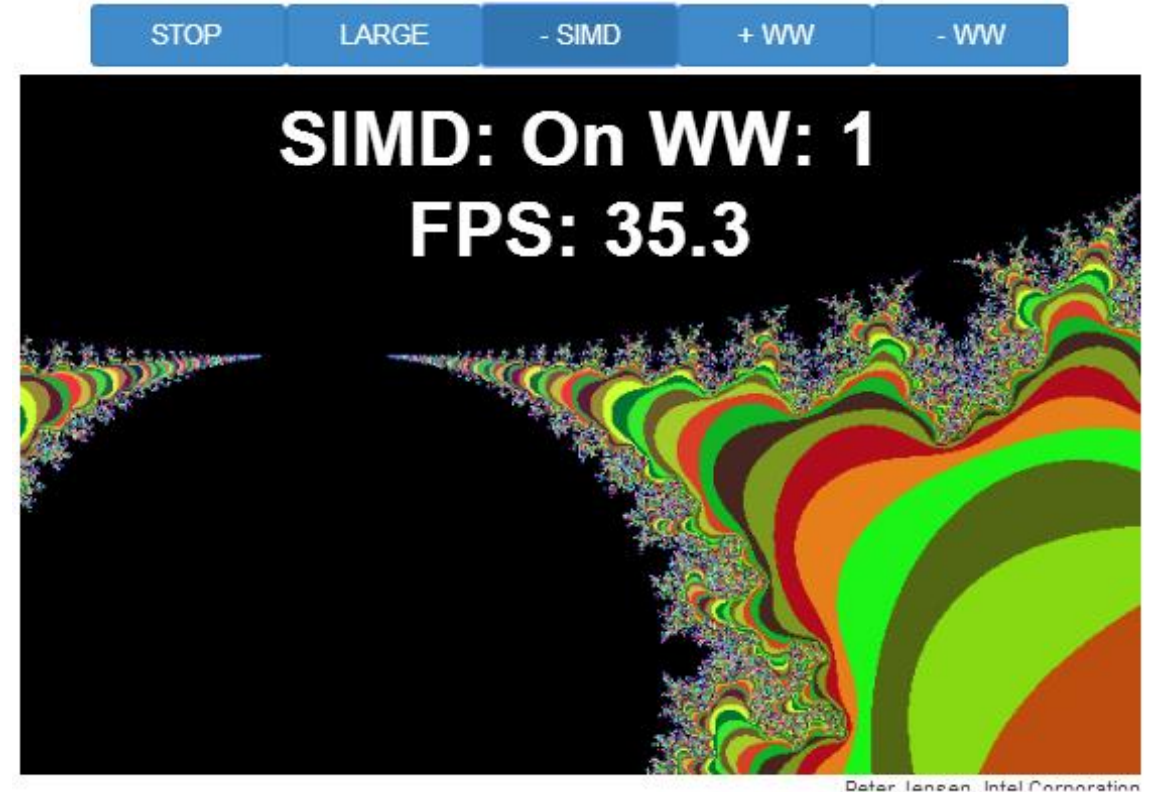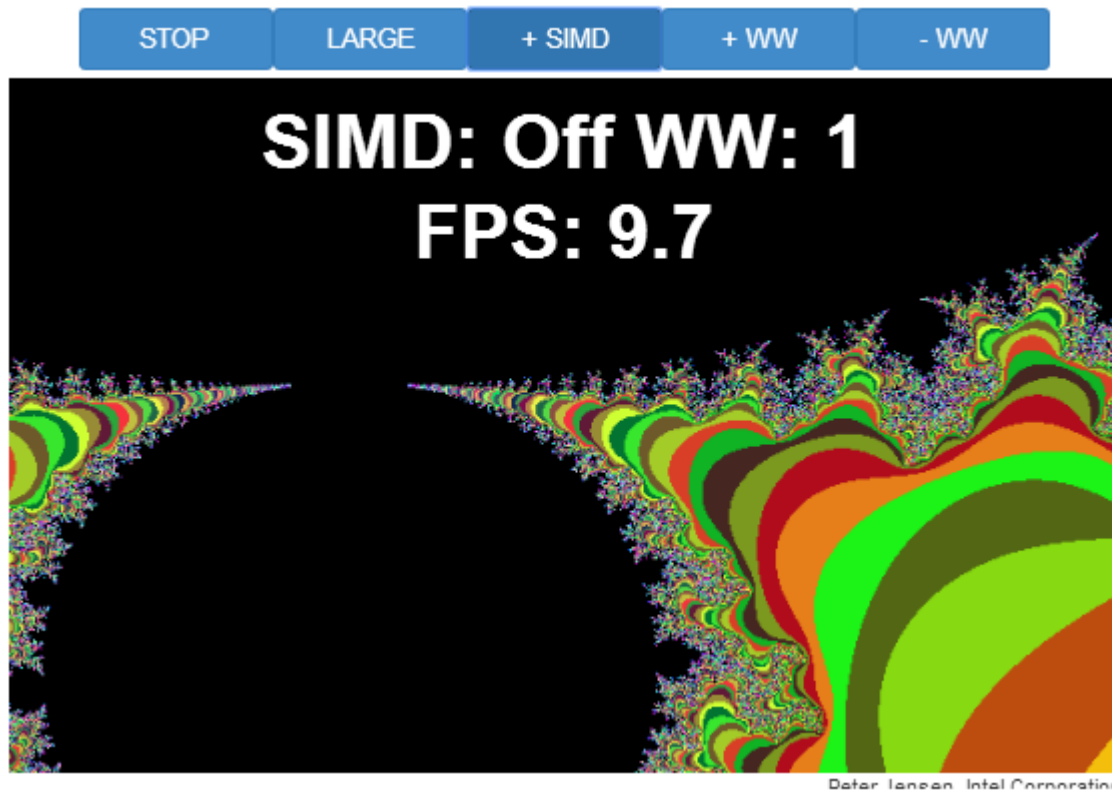
More Silicon Dedicated to Vector Processing
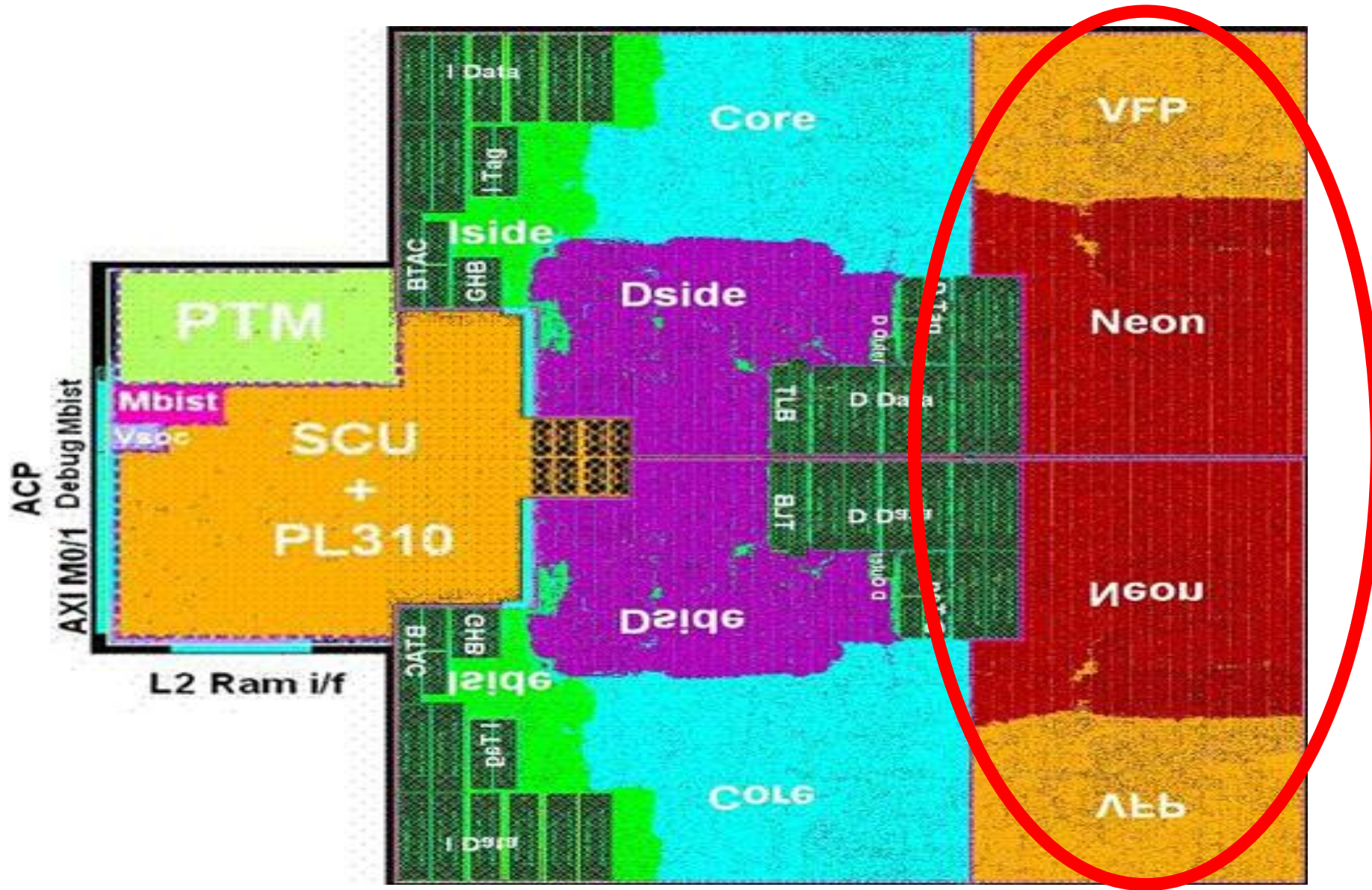
Hardware/Software Gap

- SIMD instructions are an increasingly larger portion of instruction set architectures of newer CPUs

- Currently, it's not possible to utilize these powerful instructions from JavaScript programs

SIMD.JS/Emscripten will bridge this gap

Demo - Mandelbrot

SIMD.JS/Emscripten – Background/History

- **Intel/Mozilla/Google/Microsoft/ARM collaboration!**

- Started mid-2013

- Initial polyfill spec by John McCutchan (Google's Dart VM team)

- Prototypes for Chromium, Firefox

- **Available in Intel's Crossswalk web-runtime (for hybrid HTML5 apps) TODAY!**

- Emscripten (C++ -> JavaScript compiler) now generates SIMD.JS code from LLVM vector operations and from a subset of x86 SIMD intrinsics

- **Standardization (TC39) underway for inclusion of SIMD.JS in EcmaScript 7**

SIMD.JS – Object Hierarchy
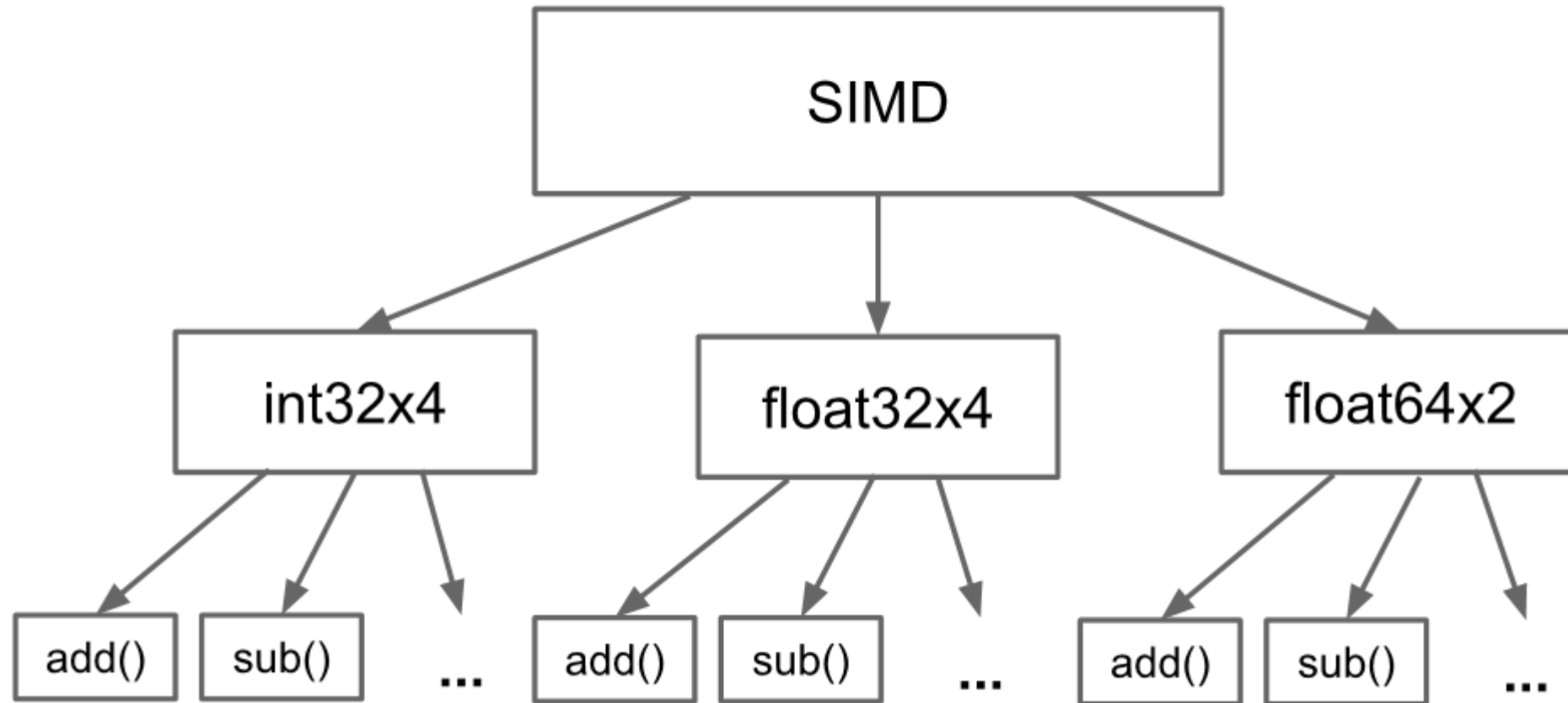
SIMD.JS – API Details

Lane accessors, mutators:

- **Accessors:** x, y, z, w

- **Mutators:** withX, withY, withZ, withW

Operators:

- **Arithmetic:** abs, neg, add, sub, mul, div, reciprocal, reciprocalSqrt, sqrt

- **Shuffle:** swizzle (1 operand), shuffle (2 operands)

- **Logical:** and, or, xor, not

- **Comparison:** equal, greaterThan, LessThan

- **Shifts:** shiftRightLogicalByScalar, shiftRightArithmeticByScalar, shiftLeftByScalar

- **Conversion:** fromInt32x4, fromInt32x4Bits, etc.

- **Min/Max:** min, minNum, max, maxNum

SIMD.JS – Example Usage - Mandelbrot

Scalar

```
//
// z(i+1) = z(i)^2 + c
// terminate when |z|^2 > 4.0
// returns 1 iteration count
//
function mandelx1 (c_re, c_im) {
  var z_re = c_re, z_im = c_im, i;


  for (i = 0; i < max_iterations; ++i) {
    var z_re2 = z_re*z_re;
    var z_im2 = z_im*z_im;

    if (z_re2 + z_im2 > 4.0) {
      // iteration has diverged
      break;
    }
    var new_re = z_re2 - z_im2;
    var new_im = 2.0 * z_re * z_im;
    z_re = c_re + new_re;
    z_im = c_im + new_im;

  }
  return i;
}
```

SIMD

```
function mandelx4(c_re4, c_im4) {
    var z_re4  = c_re4,
        z_im4  = c_im4,
        four4  = SIMD.float32x4.splat (4.0),
        two4   = SIMD.float32x4.splat (2.0),
        count4 = SIMD.int32x4.splat (0),
        one4   = SIMD.int32x4.splat (1),
        i, z_re24, z_im24, mi4, new_re4, new_im4;

    for (i = 0; i < max_iterations; ++i) {
      z_re24 = SIMD.float32x4.mul (z_re4, z_re4);
      z_im24 = SIMD.float32x4.mul (z_im4, z_im4);
      mi4 = SIMD.float32x4.greaterThan(SIMD.float32x4.add (z_re24, z_im24), four4);
      if (SIMD.int32x4.allTrue()) {
        // all 4 values have diverged
        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.JS – Focus

Initial focus on architecture overlap (128-bit vectors)

- Well defined NaN handling

- Well defined float32 -> int32 conversions

- Well defined shift handling for shift counts > 32

- Precision of reciprocalSqrt – left undefined

Architecture specific extensions are in the works, for example

- Fma (NEON, AVX)

- Vector shifts (NEON)

Emscripten - Basics

- Brainchild of Mozilla's Alon Zakai

- Compiles C/C++ to JavaScript

- Uses clang/LLVM for C/C++ front-end and optimizer framework

- Models memory with JS TypedArrays

- Generates the asm.js subset of JavaScript

- Tools available to create bindings between handwritten JS and Emscripten generated JS (webidl_binder)

- Several large C/C++ applications/games have been ported to the web platform (e.g., Unity, Unreal, box2D)

Emscripten – C/C++ -> JS Memory Modelling

- Views over the same memory (buffer) for basic C/C++ types

- C/C++ pointers used as indices to access elements of these arrays

```
var buffer = new ArrayBuffer(TOTAL_MEMORY);
HEAP8 = new Int8Array(buffer);
HEAP16 = new Int16Array(buffer);
HEAP32 = new Int32Array(buffer);
HEAPU8 = new Uint8Array(buffer);
HEAPU16 = new Uint16Array(buffer);
HEAPU32 = new Uint32Array(buffer);
HEAPF32 = new Float32Array(buffer);
HEAPF64 = new Float64Array(buffer);
```

Emscripten – Generated Code Example

```
for (uint32_t j = 0, l = length;
     j < l;
     j = j + 4) {
  sum = sum + *(a++);
}
```

```
while (1) {
  $add = $sum$04 + +HEAPF32[$a$addr$06 >> 2];
  $j$05 = $j$05 + 4 | 0;
  if (!($j$05 >>> 0 < $length >>> 0)) {
    $sum$0$lcssa = $add;
    break;
  } else {
    $a$addr$06 = $a$addr$06 + 4 | 0;
    $sum$04 = $add;
  }
}
```

- Unary '+' (+expr) used as hint to JIT compiler to indicate number

- Bitwise-Or zero (expr|0) used to indicate 32-bit signed int

- Unsigned shift right (>>>) used to indicate 32-bit unsigned int

- Addresses are byte addresses. Need to shift right by 2 to get the right index
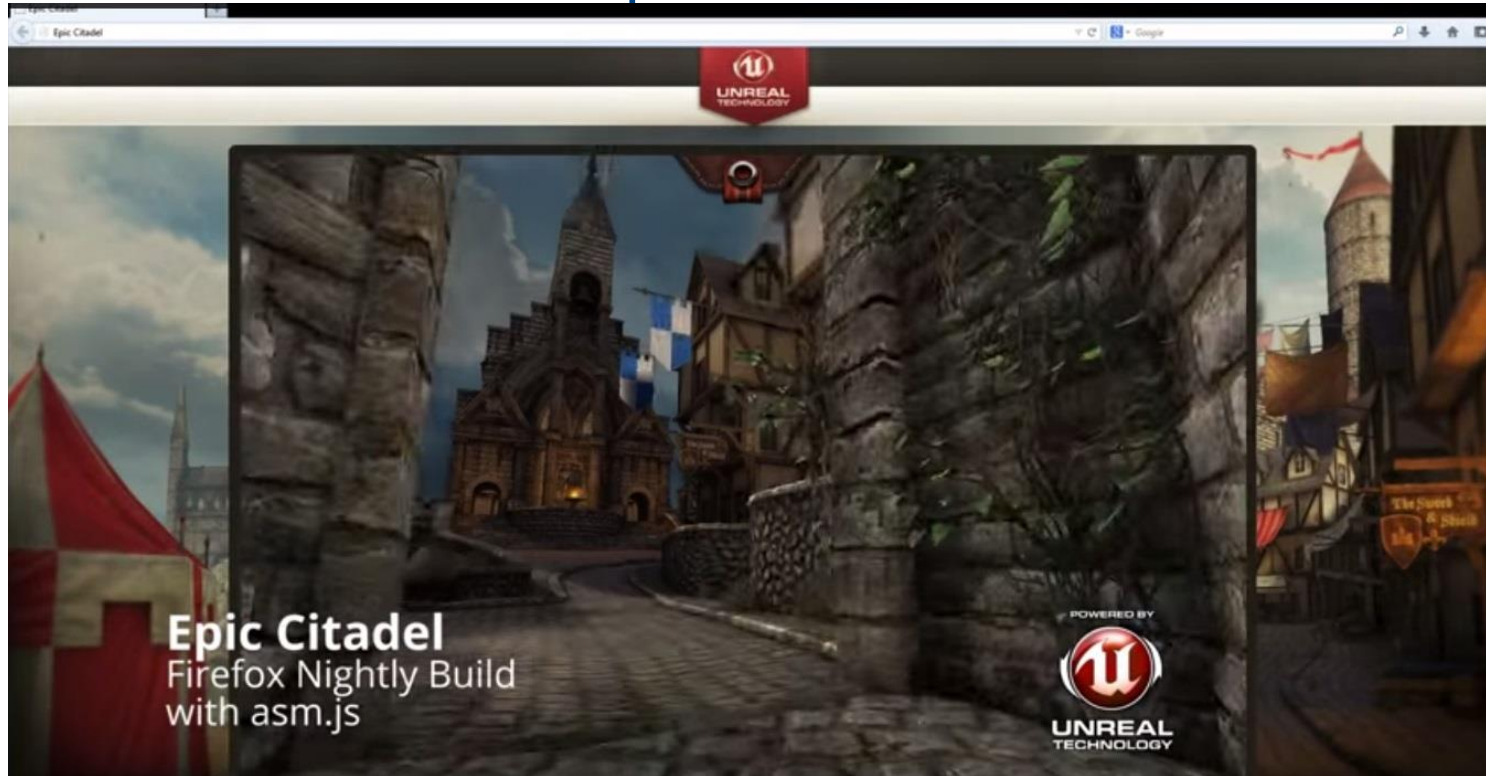
Type hints and no dynamic allocations allow JIT compilers to generate very efficient code QUICKLY!
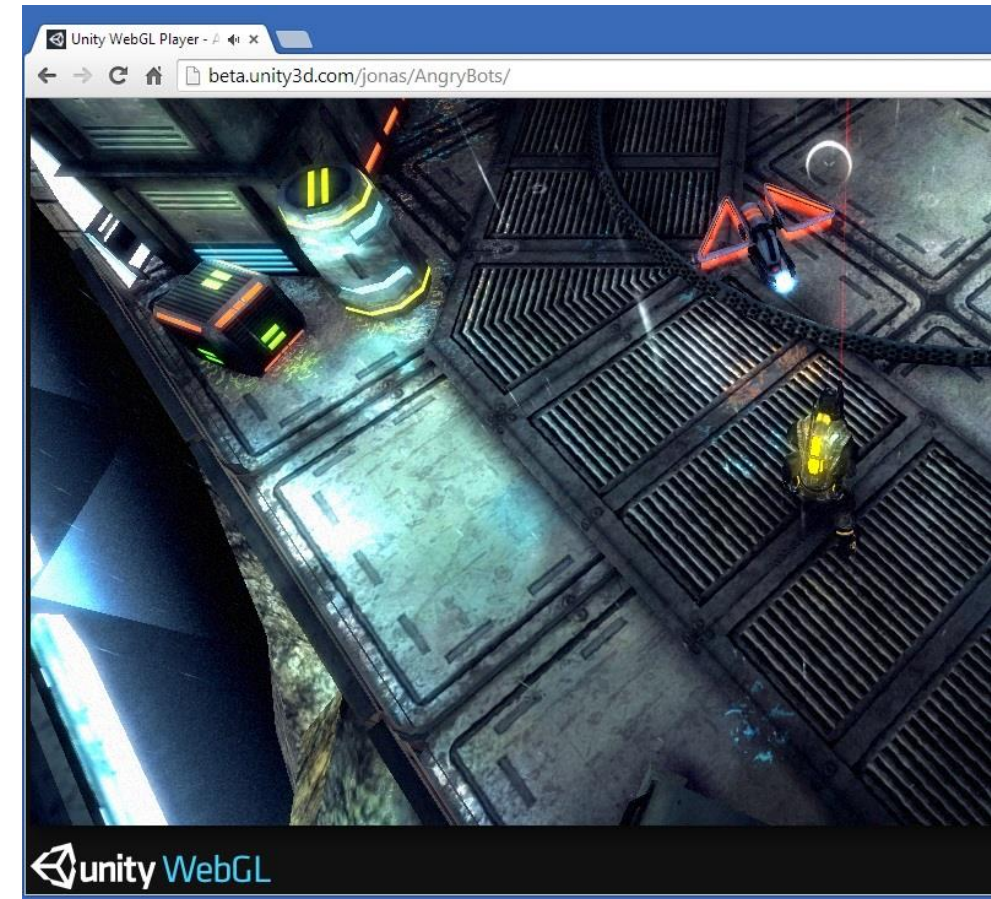
Emscripten – Showcase Uses

Epic Unreal

Unity

- Over a million lines of C++ code ported to JS
- 4 days to port!

SIMD.JS and Emscripten – A Perfect Match!

- Performance critical C/C++ code uses SIMD to get acceptable performance

    - Games, physics, image manipulation, video encoding/decoding, signal processing, etc.

- SIMD.JS enables Emscripten to fully utilize these highly optimized C/C++ code sequences

Emscripten – Compiling SIMD C/C++ Code

$ emcc –O2 –g average-intrin.c

C code with x86 intrinsics

```
float averageIntrin(float *a,
                    uint32_t length) {
__m128 sumx4 = _mm_set_ps1(0.0);
for (uint32_t j = 0, l = length;
     j < l; j = j + 4) {
  sumx4 = _mm_add_ps(
          sumx4, _mm_loadu_ps(&(a[j])));
}
float mSumx4[4];
_mm_storeu_ps(mSumx4, sumx4);
return (mSumx4[0] + mSumx4[1] +
        mSumx4[2] + mSumx4[3])/length;
}
```

asm.js code with SIMD.JS (for loop)

```
while (1) {
  $add$i = SIMD_float32x4_add(
           $sumx4$010,
           SIMD_float32x4_load(
           buffer, $a + ($j$09 << 2) | 0));
  $j$09 = $j$09 + 4 | 0;
  if (!($j$09 >>> 0 < $length >>> 0)) {
    $sumx4$0$lcssa = $add$i;
    break;
  } else $sumx4$010 = $add$i;
}
```

(intel)

Benchmarks

- Handwritten JavaScript benchmark kernels:

    - Average, Mandelbrot, MatrixMultiplication, VertexTransform, MatrixTranspose, MatrixInverse

    - Vector/Matrix math important for game/physics

    - Kernels for both scalar and SIMD implementations

    - Measure speedup (SIMD/scalar)

- Manually converted to C++

- Automatically compiled back to JavaScript with Emscripten

- JavaScript code executed with both Chromium and Firefox SIMD prototypes

- Native clang/LLVM compiler used to compile C++ code

(intel)

Benchmarks – Handwritten JavaScript

Scalar JavaScript

```
function average(n) {
  for (var i = 0; i < n; ++i) {
    var sum = 0.0;
    for (var j = 0, l = a.length; j < l; ++j) {
      sum += a[j];
    }
  }
  return sum/a.length;
}
```

SIMD JavaScript

```
function simdAverage(n) {
  for (var i = 0; i < n; ++i) {
    var sum4 = SIMD.float32x4.splat(0.0);
    for (var j = 0; j < a.length / 4; ++j) {
      sum4 = SIMD.float32x4.add(sum4, SIMD.float32x4.load(a, j << 2));
    }
  }
  return (sum4.x + sum4.y + sum4.z + sum4.w)/a.length;
}
```

Benchmarks – Handwritten C++

Scalar C++

```cpp
static float nonSimdAverageKernel32() {
  float sum = 0.0;
  for (uint32_t j = 0, l = length; j < l; ++j) {
    sum += a[j];
  }
  return sum/length;
}
```

SIMD C++

```cpp
static float simdAverageKernel() {
  __m128 sumx4 = _mm_set_ps1(0.0);
  for (uint32_t j = 0, l = length; j < l; j = j + 4) {
    sumx4 = _mm_add_ps(sumx4, _mm_loadu_ps(&(a[j])));
  }
  Base::Lanes<__m128, float> lanes(sumx4);
  return (lanes.x() + lanes.y() + lanes.z() + lanes.w())/length;
}
```
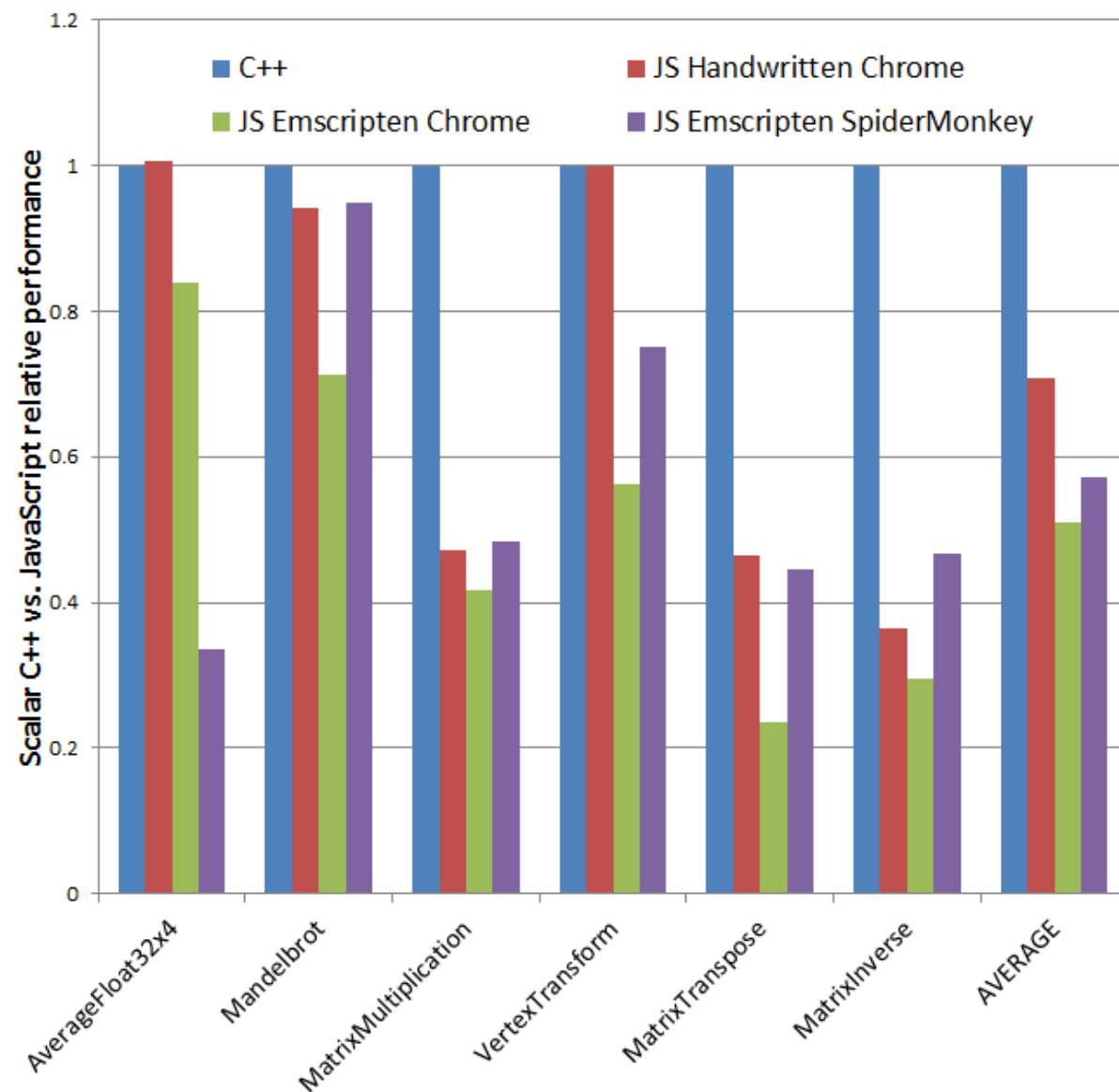
Benchmark Results – SIMD vs. Scalar Speedups
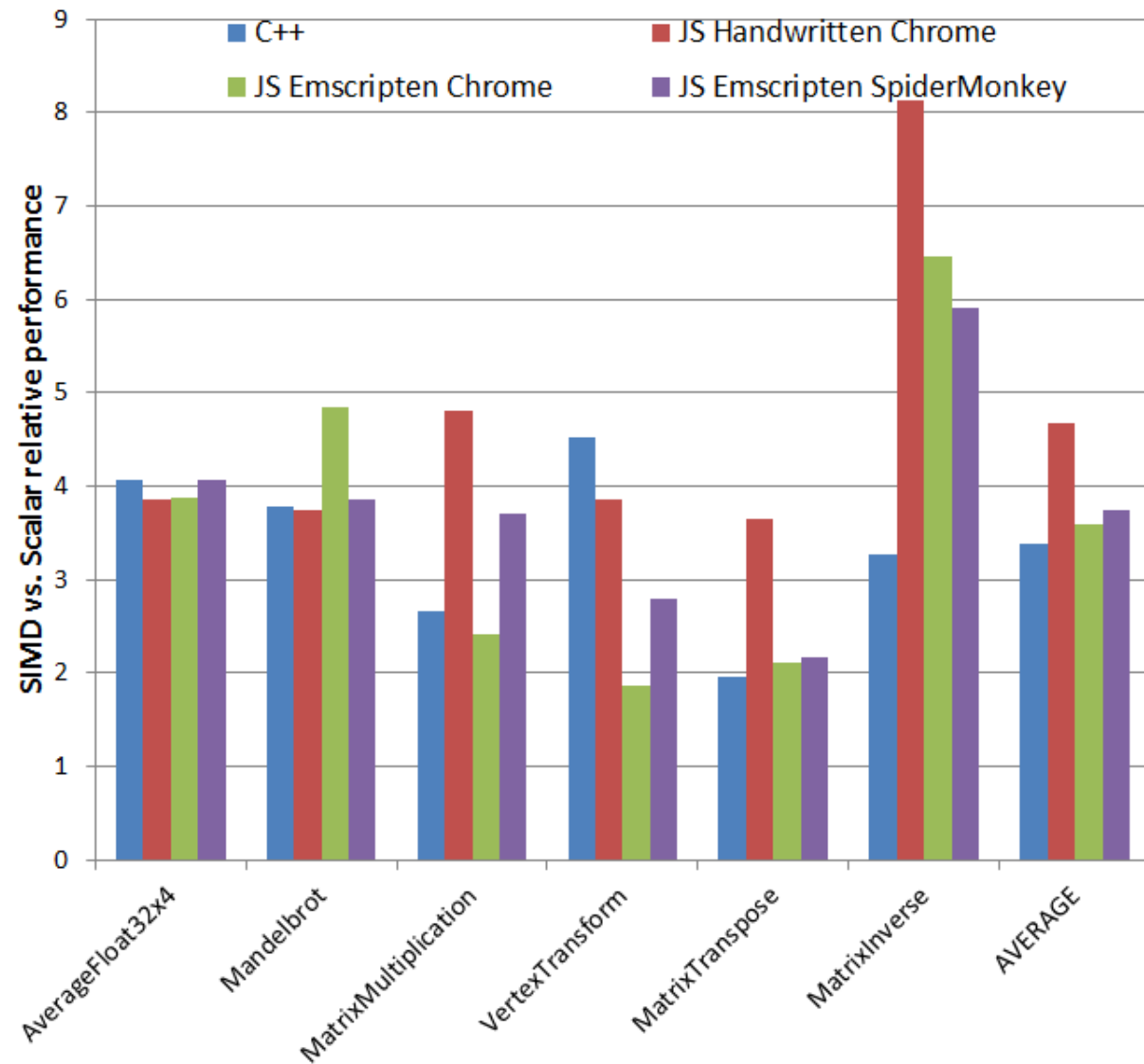
Observations:
- **Average speedups are in the expected ~4x range**
- Higher speedups for 'JS Handwritten Chrome' is due to slow scalar kernel (64-bit FP operations
- Super linear speedups for MatrixInverse most likely due to slower scalar kernel as well.

Benchmark Results – Scalar C++ vs. Scalar JS

Observations:
- **Average JS performance is roughly 60% of native C++**
- Handwritten JS code is slightly faster than Emscripten generated JS
- SpiderMonkey/OdinMonkey is slightly faster than Chromium on Emscripten generated JS
- Chromium has 'slow JIT' overhead that OdinMonkey doesn't

Benchmark Results – SIMD C++ vs. SIMD JS
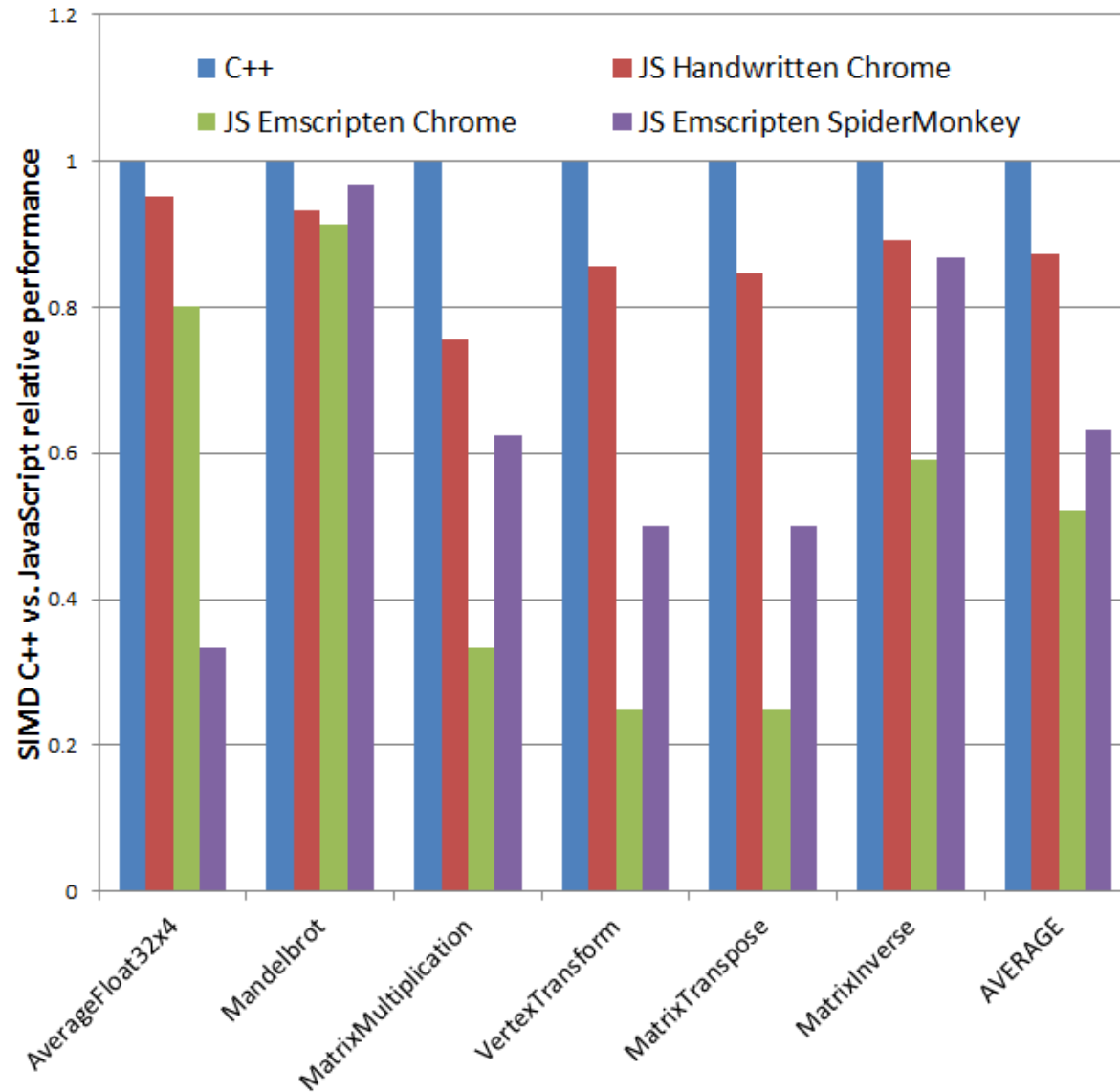
Observations:
- Handwritten JS performance is ~85% of native C++ on Chromium!
- Emscripten generated JS performance is ~60% of native C++ on both Chromium and OdinMonkey
- C++/JS performance for SIMD code is roughly the same as it is for Scalar code

Summary

SIMD.JS bridges the hardware/software gap for JavaScript programs

SIMD.JS makes ~4x speedup of performance critical code possible

Emscripten now compiles SIMD C++ vector code

The performance gap between native C/C++ code and JavaScript code keeps getting smaller

HTML5/JavaScript is rapidly becoming a viable cross platform

THANK YOU

References

SIMD.JS polyfill/spec and handwritten JS benchmarks:

- https://github.com/johnmccutchan/ecmascript_simd

Emscripten:

- https://github.com/kripken/emscripten/wiki

C++ benchmarks:

- https://github.com/PeterJensen/benchcpp

This Presentation:

- https://github.com/PeterJensen/wpmvp2015