SIMD in JavaScript* via C++ and Emscripten*

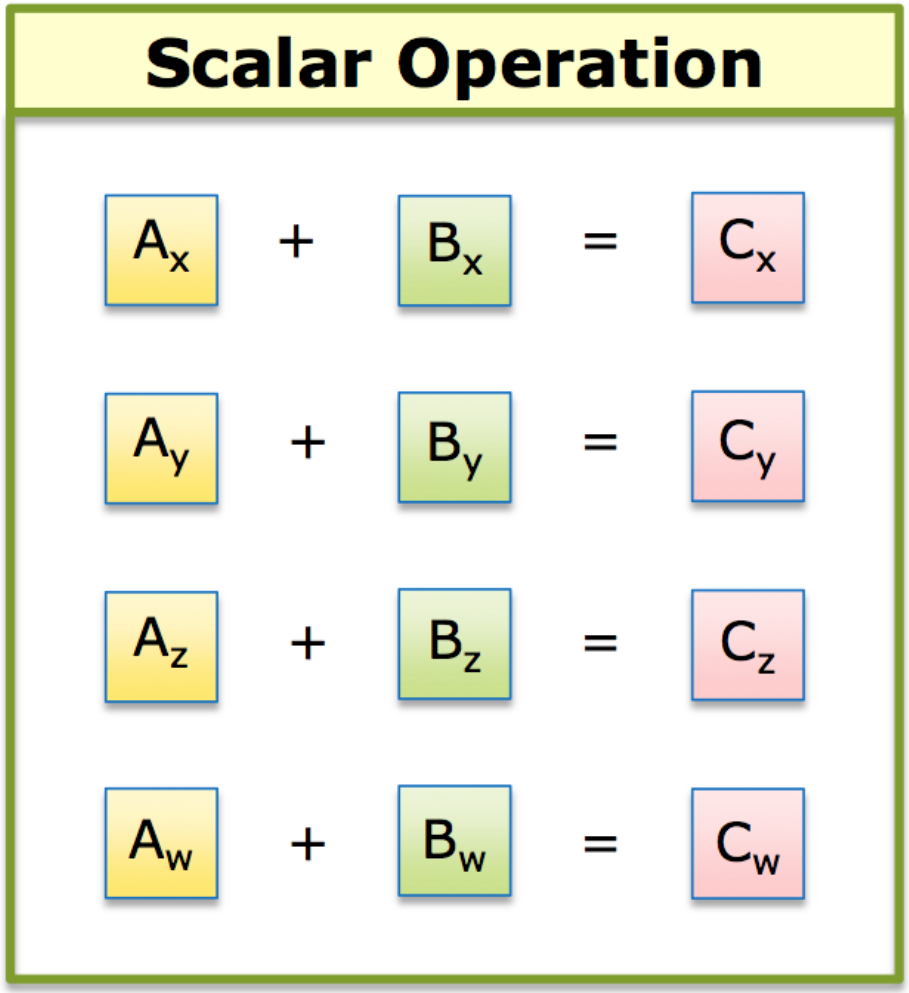
February 8th, 2015 - Workshop on Programming Models for SIMD/Vector Processing

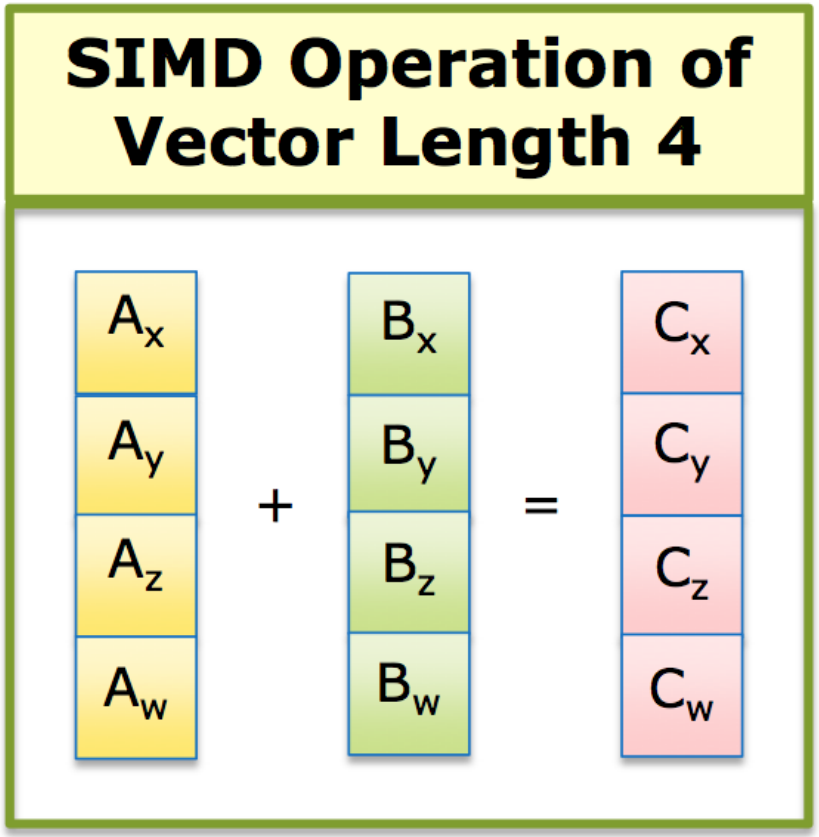
Peter Jensen, Intel Corporation
Ivan Jibaja, Intel Corporation
Ningxin Hu, Intel Corporation
Dan Gohman, Mozilla*
John McCutchan, Google*

Agenda

- Motivation
- Background/History
- SIMD.JS
- Emscripten*
- Compiling SIMD C++ code to SIMD.JS JavaScript* code
- Benchmark Results
- Summary

SIMD: Single Instruction, Multiple Data

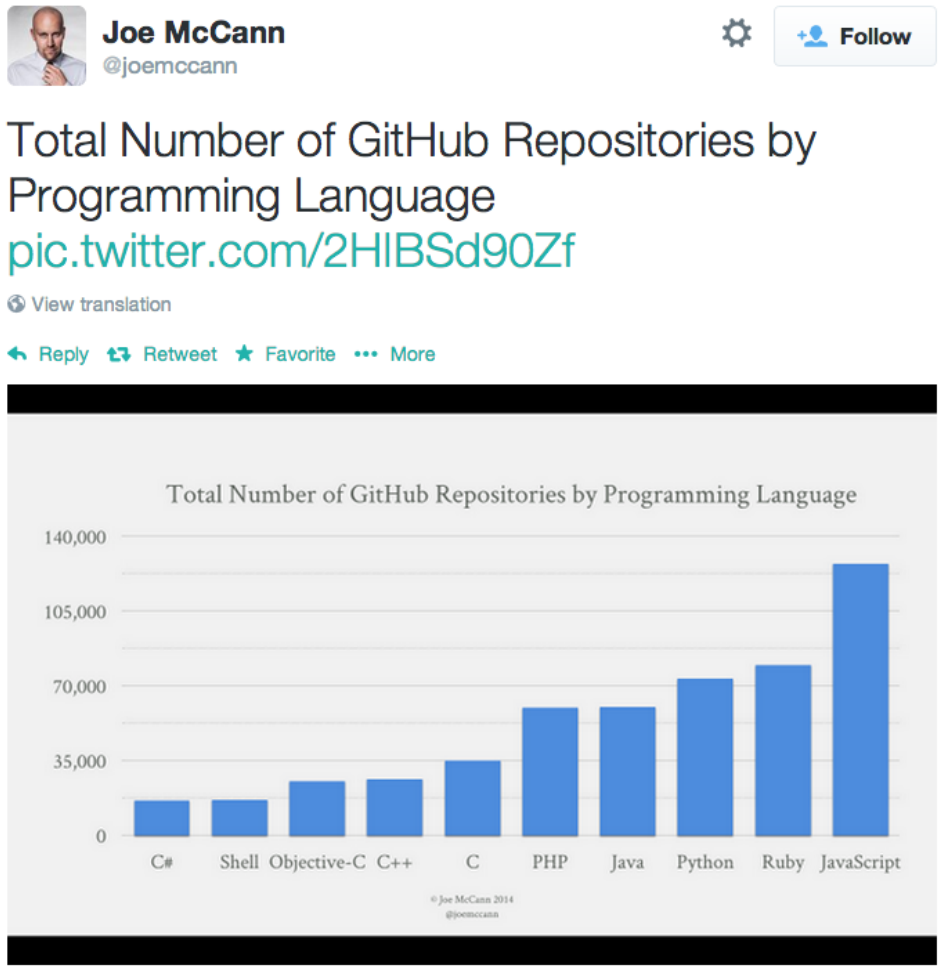




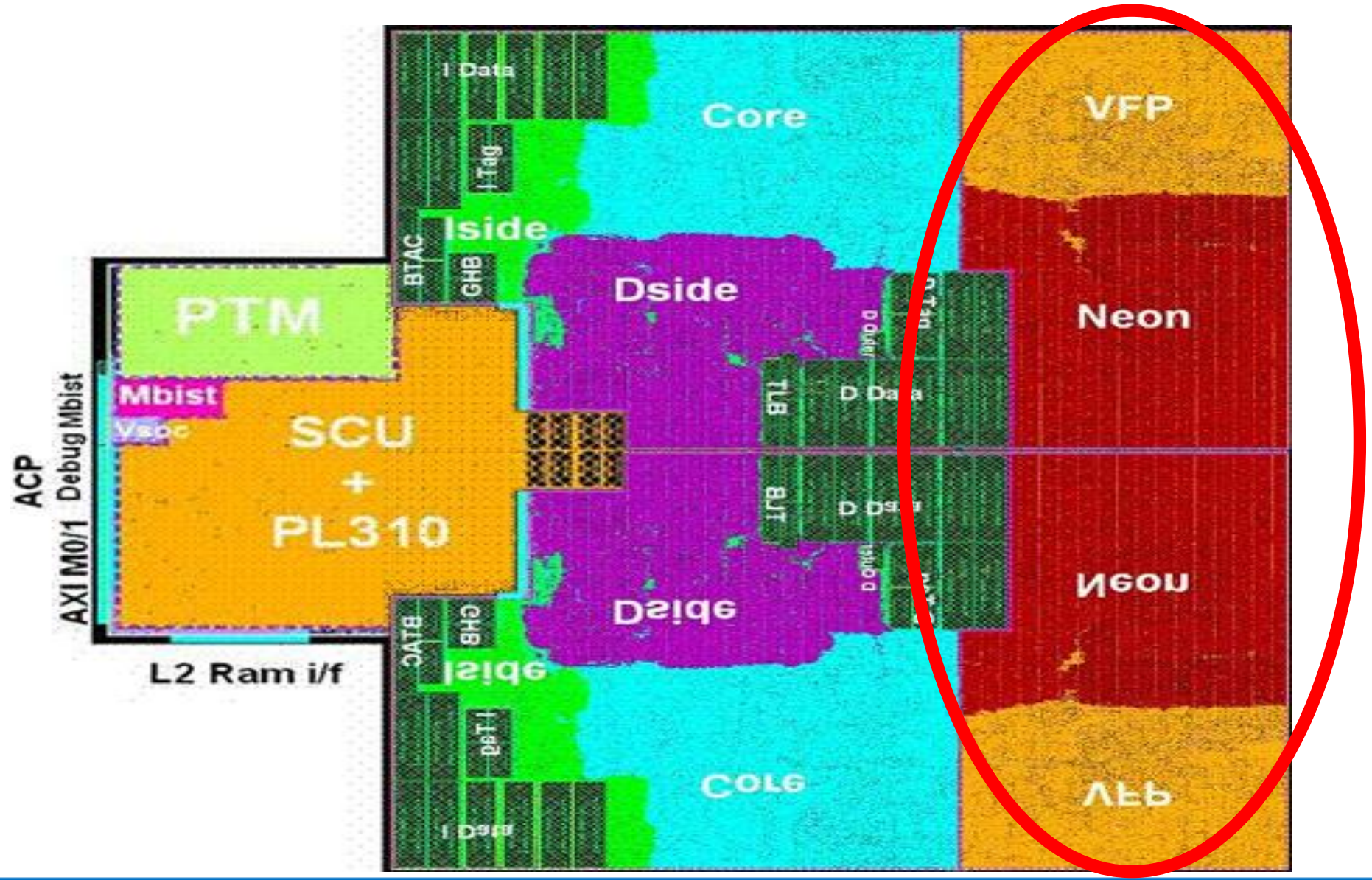
Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

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, Office365*, …)
- Server side logic via node.js*/io.js*



More Silicon Dedicated to Vector Processing

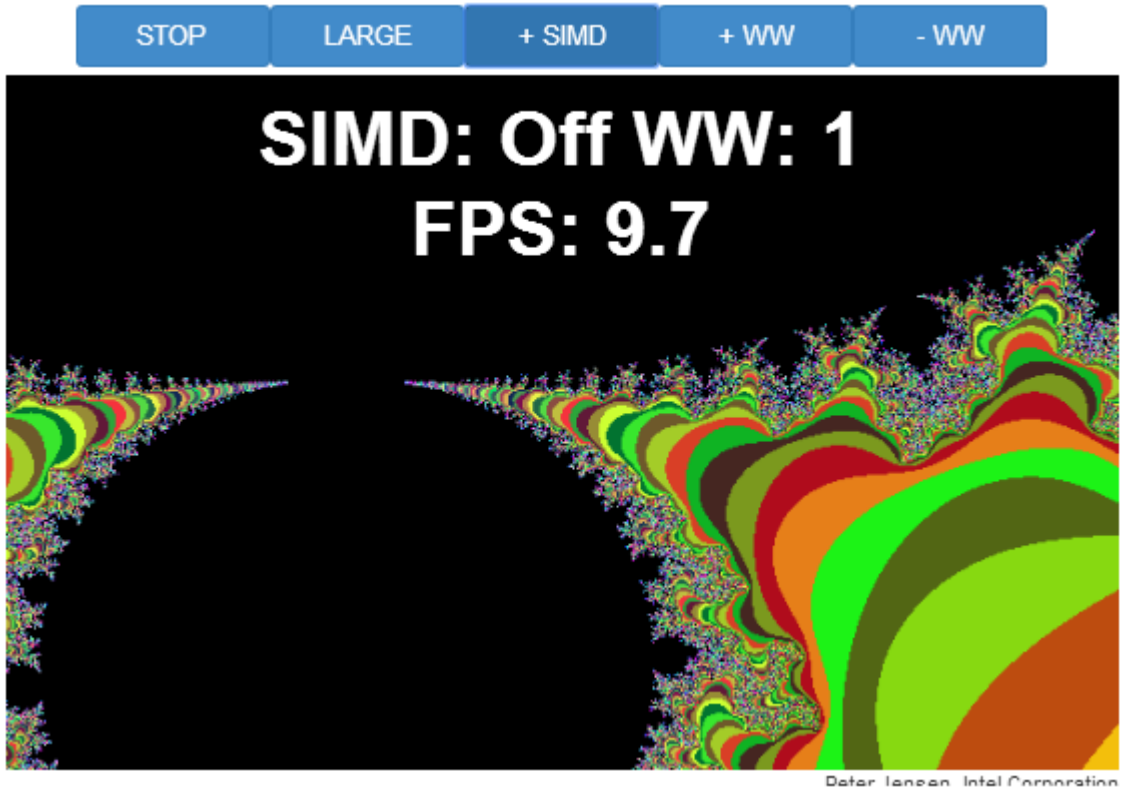


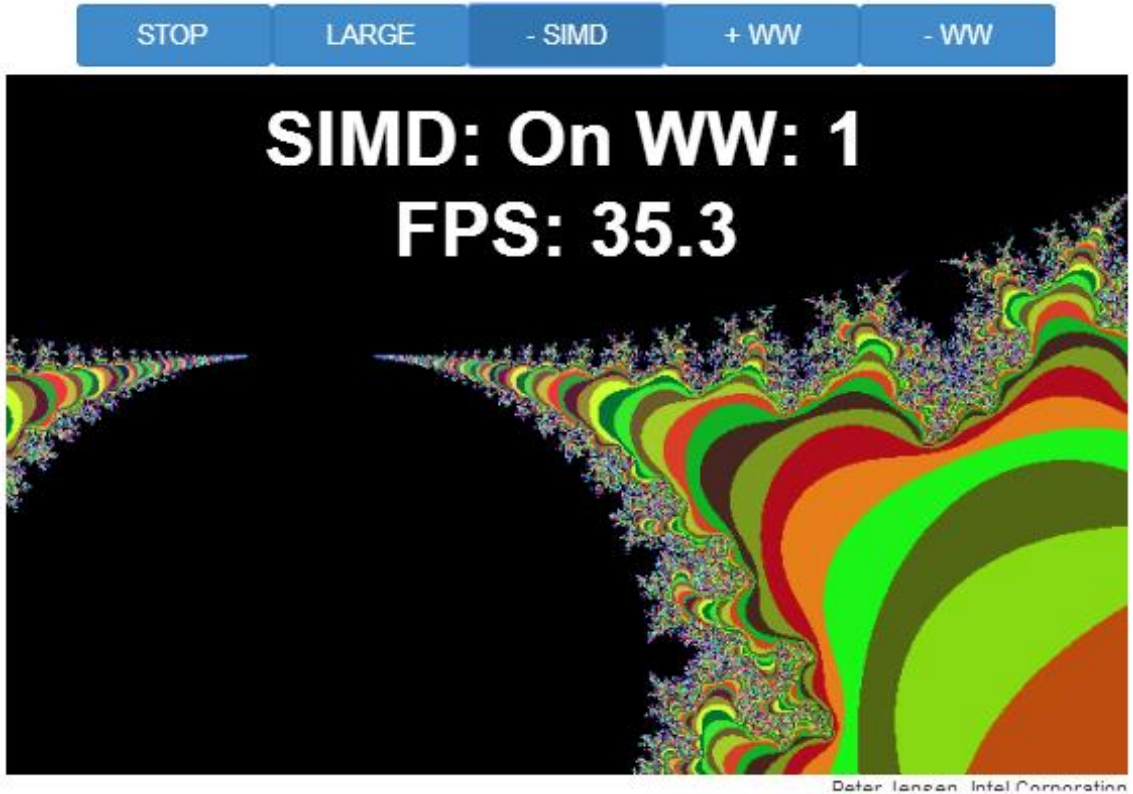
Hardware/Software Gap

- 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

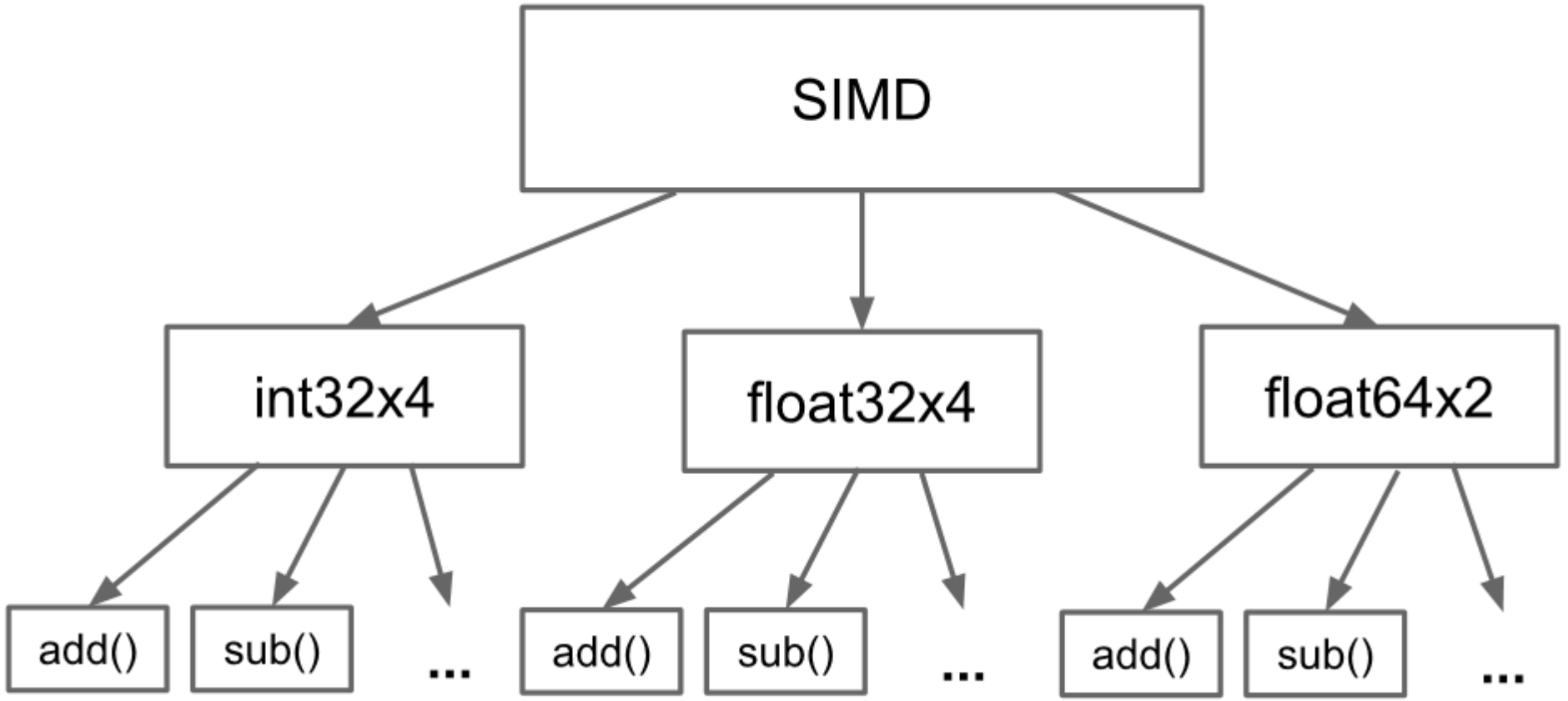




SIMD.JS/Emscripten – Background/History

- **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 being discussed, 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++);
}
```

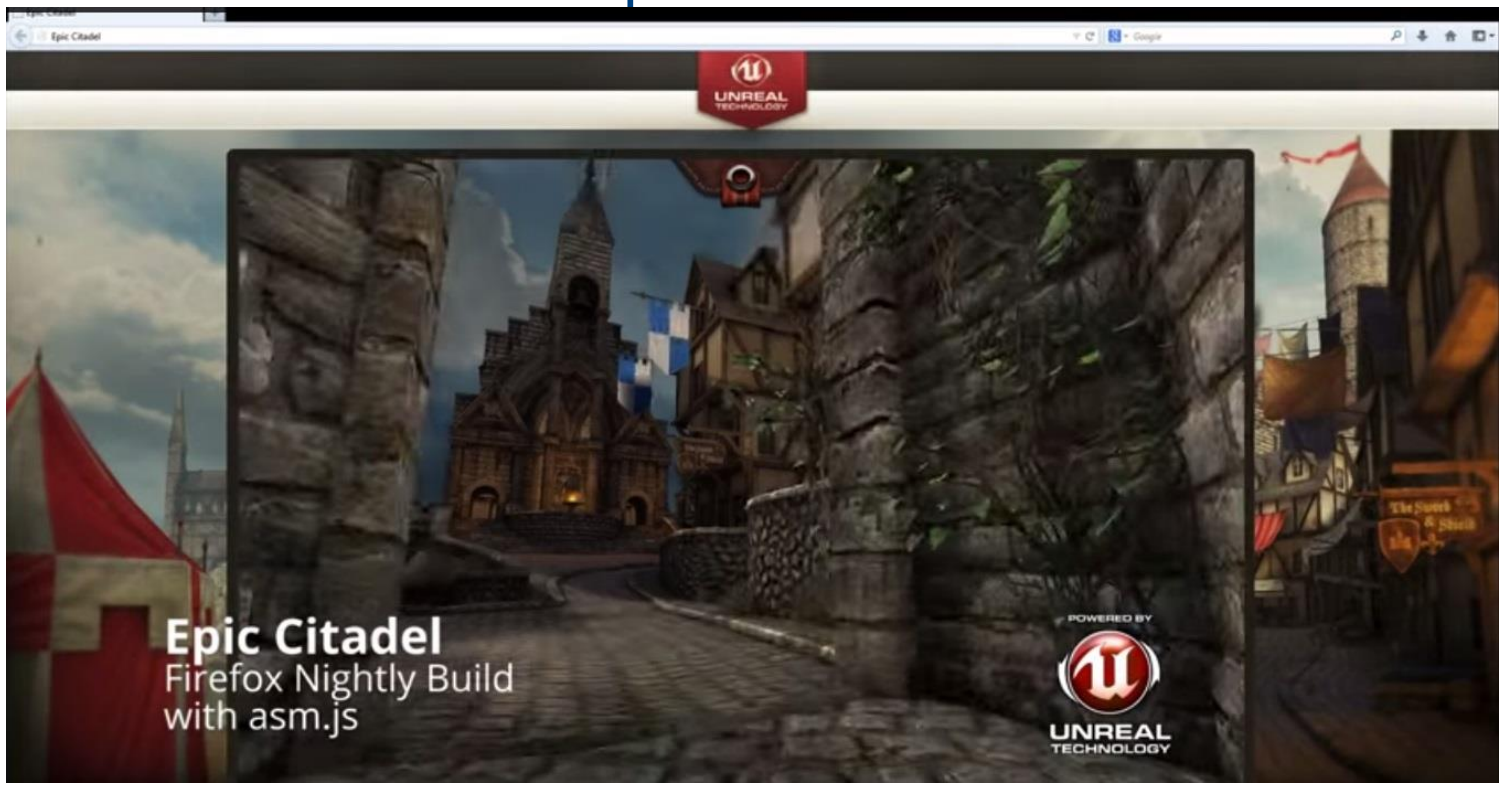
- Unary '+' (+expr) used as hint to JIT compiler to indicate number
- Bitwise-Or zero (expr|0) used to indicate 32-bit signed int
- Unsigned shift right (>>>) used to indicate 32-bit unsigned int
- Addresses are byte addresses. Need to shift right by 2 to get the right index

```
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;
 }
}
```

Type hints and no dynamic allocations allow JIT compilers to generate very efficient code QUICKLY!

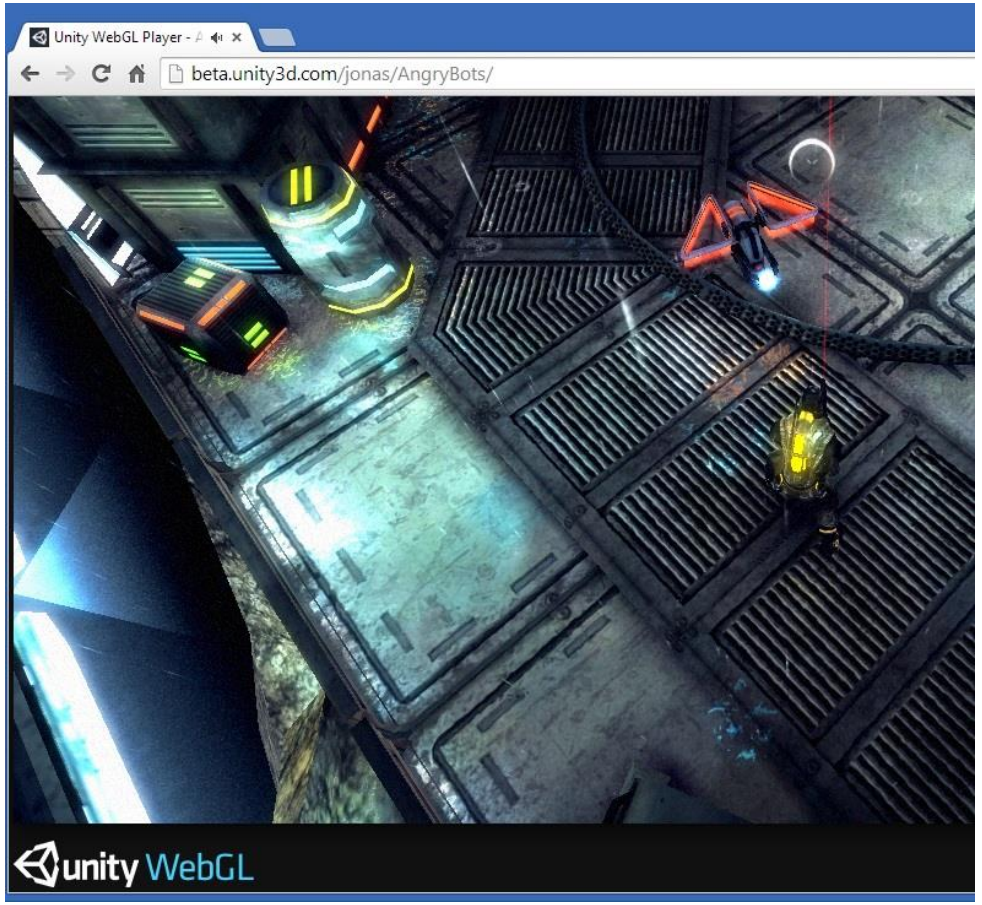
Emscripten* – Showcase Uses

Epic Unreal



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

Unity



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;
}
```

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

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++

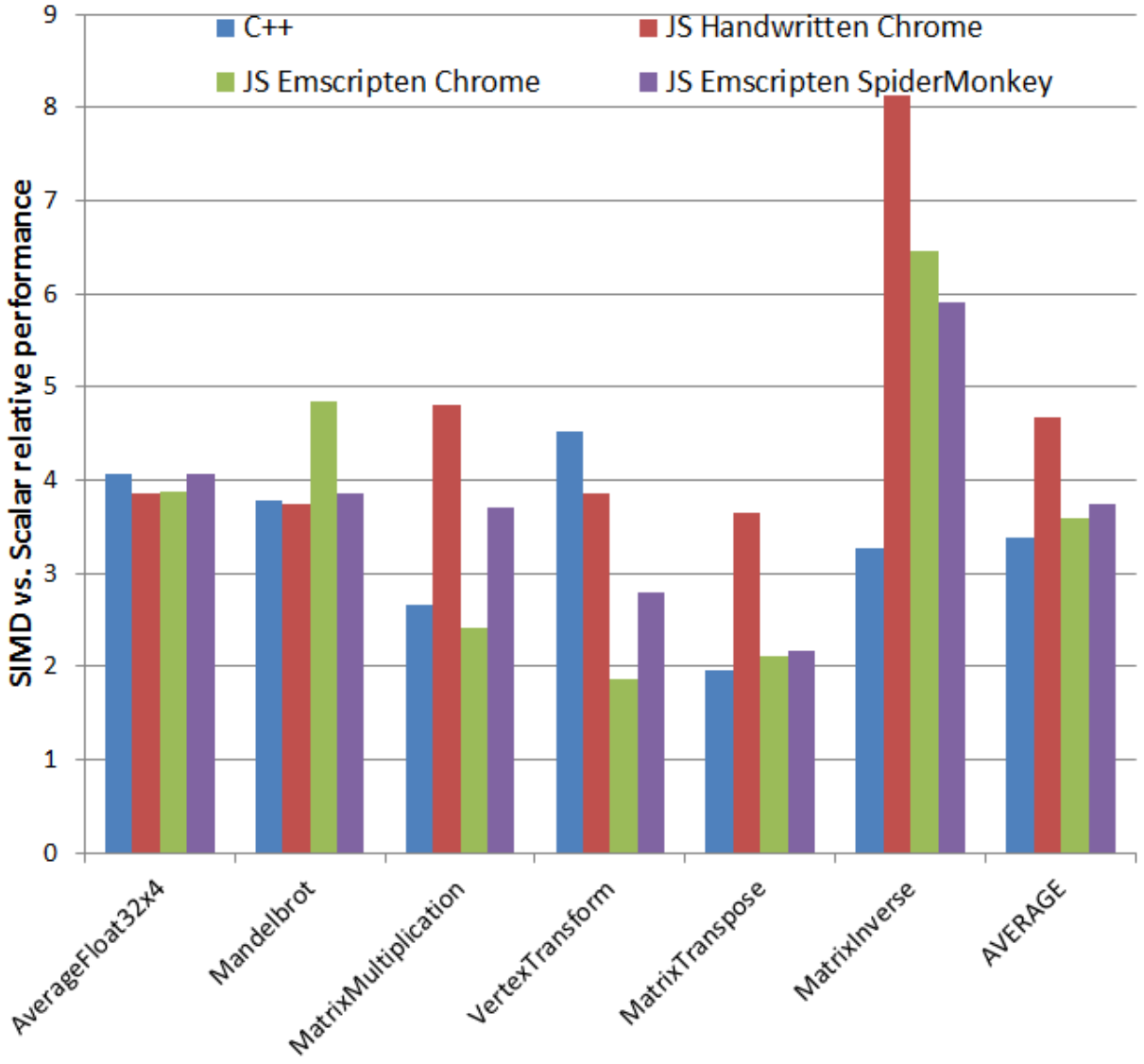
Scalar C++

```
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++

```
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;
}
```

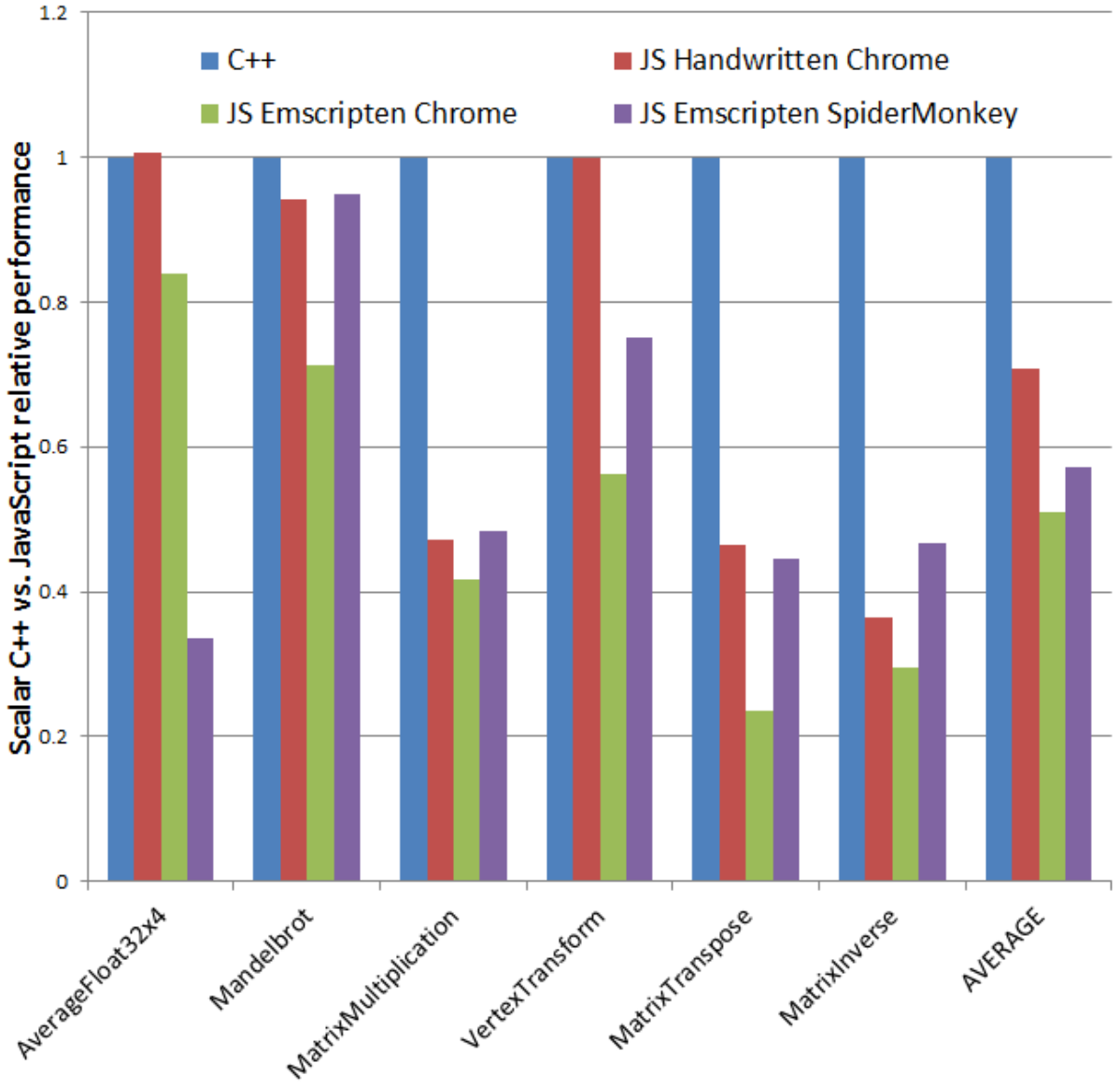
Benchmark Results – SIMD vs. Scalar Speedups



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.

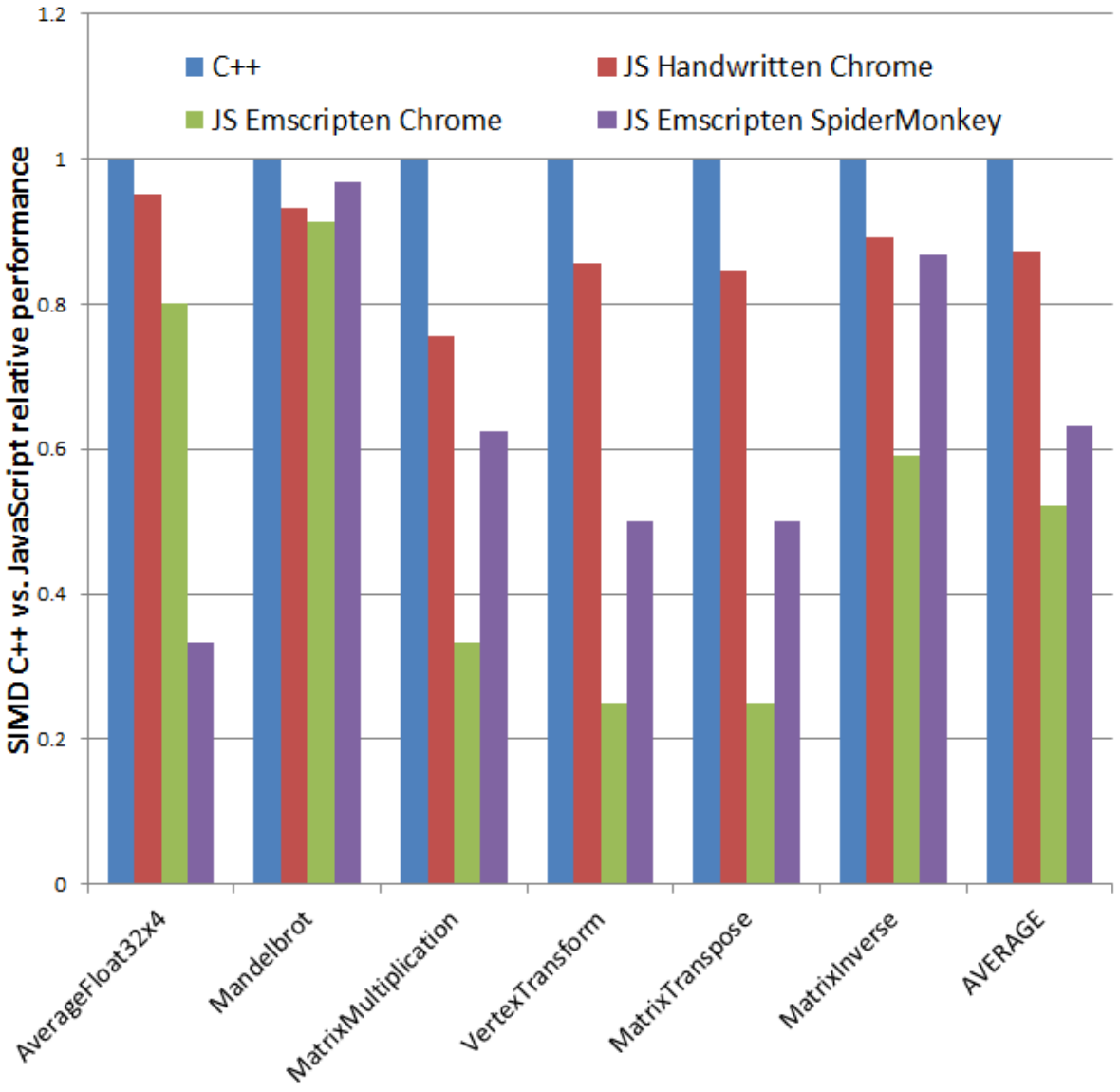
Benchmark Results – Scalar C++ vs. Scalar JS



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 becomes an increasingly capable 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 and accompanying paper (contact info in paper):

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

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

River Trail, Nehalem, and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.

Intel, Xeon, VTune, Atom, Core, Xeon Phi, Look Inside and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Legal Disclaimer

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

- Software Source Code Disclaimer: Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Risk Factors

The above statements and any others in this document that refer to plans and expectations for the third quarter, the year and the future are forward-looking statements that involve a number of risks and uncertainties. Words such as “anticipates,” “expects,” “intends,” “plans,” “believes,” “seeks,” “estimates,” “may,” “will,” “should” and their variations identify forward-looking statements. Statements that refer to or are based on projections, uncertain events or assumptions also identify forward-looking statements. Many factors could affect Intel’s actual results, and variances from Intel’s current expectations regarding such factors could cause actual results to differ materially from those expressed in these forward-looking statements. Intel presently considers the following to be the important factors that could cause actual results to differ materially from the company’s expectations. Demand could be different from Intel's expectations due to factors including changes in business and economic conditions; customer acceptance of Intel’s and competitors’ products; supply constraints and other disruptions affecting customers; changes in customer order patterns including order cancellations; and changes in the level of inventory at customers. Uncertainty in global economic and financial conditions poses a risk that consumers and businesses may defer purchases in response to negative financial events, which could negatively affect product demand and other related matters. Intel operates in intensely competitive industries that are characterized by a high percentage of costs that are fixed or difficult to reduce in the short term and product demand that is highly variable and difficult to forecast. Revenue and the gross margin percentage are affected by the timing of Intel product introductions and the demand for and market acceptance of Intel's products; actions taken by Intel's competitors, including product offerings and introductions, marketing programs and pricing pressures and Intel’s response to such actions; and Intel’s ability to respond quickly to technological developments and to incorporate new features into its products. The gross margin percentage could vary significantly from expectations based on capacity utilization; variations in inventory valuation, including variations related to the timing of qualifying products for sale; changes in revenue levels; segment product mix; the timing and execution of the manufacturing ramp and associated costs; start-up costs; excess or obsolete inventory; changes in unit costs; defects or disruptions in the supply of materials or resources; product manufacturing quality/yields; and impairments of long-lived assets, including manufacturing, assembly/test and intangible assets. Intel's results could be affected by adverse economic, social, political and physical/infrastructure conditions in countries where Intel, its customers or its suppliers operate, including military conflict and other security risks, natural disasters, infrastructure disruptions, health concerns and fluctuations in currency exchange rates. Expenses, particularly certain marketing and compensation expenses, as well as restructuring and asset impairment charges, vary depending on the level of demand for Intel's products and the level of revenue and profits. Intel’s results could be affected by the timing of closing of acquisitions and divestitures. Intel's results could be affected by adverse effects associated with product defects and errata (deviations from published specifications), and by litigation or regulatory matters involving intellectual property, stockholder, consumer, antitrust, disclosure and other issues, such as the litigation and regulatory matters described in Intel's SEC reports. An unfavorable ruling could include monetary damages or an injunction prohibiting Intel from manufacturing or selling one or more products, precluding particular business practices, impacting Intel’s ability to design its products, or requiring other remedies such as compulsory licensing of intellectual property. A detailed discussion of these and other factors that could affect Intel’s results is included in Intel’s SEC filings, including the company’s most recent reports on Form 10-Q, Form 10-K and earnings release.