SIMD in JavaScript via C++ and Emscripten

Peter Jensen
Intel Corporation
peter.jensen@intel.com

Ivan Jibaja
Intel Corporation
ivan.jibaja@intel.com

Ningxin Hu
Intel Corporation
ningxin.hu@intel.com

Dan Gohman
Mozilla
sunfish@mozilla.com

John McCutchan
Google Inc.
johnmccutchan@google.com

1. Abstract

Emscripten, Mozilla's C/C++ to JavaScript compiler, can be used to port existing native C/C++ applications to the Web platform. When paired with a fast JavaScript engine, the applications run at near-native speeds. However, compute intensive C/C++ applications, such as games and media-processing, making use of SIMD intrinsics or gcc style vector code cannot achieve near-native speed, because JavaScript does not have language support for expressing SIMD operations. This paper presents SIMD.JS, an evolving new JavaScript standard, enabling SIMD operations at the JavaScript language level, and extensions to Emscripten, enabling compilation of C++ programs that make use of SIMD intrinsics or gcc style vector code. An extensive subset of available C++ SIMD x86 intrinsics is supported. Using Emscripten, we show how C/C++ SIMD programs are compiled into JavaScript resulting in equivalent speedups compared to scalar code. The new powerful combination of SIMD.JS enabled JavaScript engines and Emscripten's newly added ability to utilize the SIMD.JS primitives for it's code generation will allow a new set of compute intensive C/C++ applications to be ported to the Web platform, thereby enabling the next leap in performance on the web platform.

2. Introduction

Games, multimedia, image and video processing, and other compute intensive applications are being ported to JavaScript and made available on the Web. A popular approach for porting existing applications, originally written in C/C++, to the web, is to use Mozilla's Emscripten C++ to JavaScript compiler. Emscripten compiles C/C++ source code into a subset of JavaScript, known as asm.js. Modern JavaScript engines are able to execute the asm.js JavaScript subset at near-native speeds without the use of any plugins. Historically, plugins have been the source of many security and instability issues in web browsers, so avoiding those is highly desirable.

[Copyright notice will appear here once 'preprint' option is removed.]

Typically, compute intensive C/C++ applications will make use of SIMD/vector code to achieve the desired performance. However, because JavaScript does not have an equivalent way of expressing SIMD operations, the JavaScript code resulting from compiling the C/C++ applications via Emscripten is missing out on potentially significant performance gains.

With the recent introduction of SIMD.JS; a new JavaScript language proposal, it is now possible to use SIMD JavaScript primitives to improve performance of critical code, thus making better use of the SIMD hardware's full potential.

This paper explores the implementation and evaluation of Emscripten's ability to generate SIMD.JS code from its native C/C++ intrinsics counterparts. We present performance data for a set of SIMD/vector benchmarks written in C++ and an equivalent set of benchmarks handwritten in JavaScript. The SIMD vs. scalar speedups for the natively compiled C++ benchmarks is compared to the speedups for the handwritten JavaScript benchmarks as well as for the benchmarks compiled into JavaScript from the C++ benchmarks. Google's V8 and Mozilla's SpiderMonkey JavaScript engines are used for JavaScript execution.

3. SIMD.JS

SIMD is short for Single Instruction, Multiple Data. It refers to CPU instruction level data parallelism. Most modern CPUs have a significant portion of their available instructions dedicated to operating on data in parallel. Typically, those instructions will perform the same operation on elements in short vectors, e.g. vectors of length 4, 8, or 16. Use of these instructions leads to increased performance, because more data processing is achieved with fewer instructions executed, and fewer instructions also means power savings, which is of outmost importance on mobile battery powered devices. Figure 1 shows how four scalar additions are combined into a single operation.

JavaScript is quickly emerging as one of the most popular languages among software developers. It was originally used for simple web page scripting for creating interactive web pages. Around 2008, very efficient and high performance JavaScript engines emerged (e.g., Firefox's TraceMonkey [8] and Chrome's V8 engines). Since then, JavaScript has become a viable language for things beyond just basic web page interactivity, as witnessed by it's use in large web based applications, such as office applications; e-mail, document processing, etc. Also, large games, which were previously standalone, natively compiled programs, have been ported to JavaScript to run within the browser environment. More recently, JavaScript has been adopted as a server side scripting language (node.js), and lately, JavaScript has found it's way to the mobile

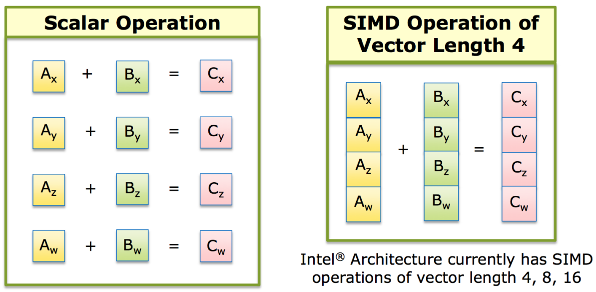

Figure 1. Replacing four scalar additions with one SIMD addition

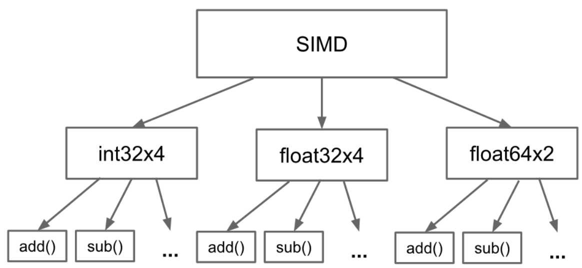

Figure 2. SIMD.JS object hierarchy

platform as a language that offers better portability between the different mobile platforms without sacrificing performance and features. For example, platform sensors (location, accelerometers, etc) are accessible from JavaScript via W3C APIs.

Even with the past 7 years of JavaScript performance advances, the desire for better performing JavaScript engines has not lessened, quite the contrary. It's a spiral that keeps on going; better performance leads to more uses, more uses require better performance. Specifically, software that uses data parallelism to achieve adequate performance have, so far, been restricted to natively compiled languages, such as C++, because such languages offer ways of utilizing the SIMD instructions available in modern CPUs. JavaScript has only one number type, Number, which is an IEEE-754 floating point number, and JavaScript offers no abstraction primitives for writing algorithms utilizing data paralellism, so it's imperative that this shortcoming is dealt with, such that the next leap in JavaScript performance is made possible. This is what the SIMD.JS proposal addresses.

SIMD.JS is an emerging standard developed collaboratively by Intel, ARM, Mozilla, Google, and Microsoft. It provides low level data types and operations that map well onto the available SIMD instructions of the underlying hardware. Currently, the defined data types and operations are a representative and useful overlap between SIMD types and operations available in most modern CPUs.

The SIMD.JS proposal is structured as an object hierarchy, with SIMD being the top level global object. The immediate properties of the SIMD object reflect the data types; `int32x4`, `float32x4`, and `float64x2`. The operations are methods declared as properties on the data type properties as outlined in Figure 2, which shows a portion of the object hierarchy.

We've modelled the semantics of the SIMD types and operations as a polyfill [2]. This allows programmers to experiment without using a JavaScript engine that natively supports SIMD.JS. The polyfill also serves as documentation for the semantics and interfaces. It will also reflect the current state of the proposal. The

```
function average(data) {
  var sum = SIMD.float32x4.splat(0.0);
  for (var i = 0, l = data.length; i < l; i = i+4) {
    sum = SIMD.float32x4.add(
      sum, SIMD.float32x4.load(data, i));
  }
  var total = sum.x + sum.y + sum.z + sum.w;
  return total/data.length;
}
```

Figure 3. SIMD JavaScript code for finding the average of an array of numbers

```
1 371 cmp eax,edx
2 373 jnl 440 (270629B8)
3 379 cmp esp,[0x645e9c]
4 385 jc 445 (270629BD)
5 391 mov esi,eax
6 393 shl esi,1
7 396 add esi,0x10
8 399 mov ebx,ecx
9 401 shl ebx,1
10 404 cmp esi,ebx
11 406 ja 499 (270629F3)
12 412 movups xmm2,[edi+eax*2]
13 416 movaps xmm3,xmm1
14 419 addps xmm3,xmm2
15 422 mov ebx,eax
16 424 add ebx,0x8
17 427 jo 504 (270629F8)
18 433 movaps xmm1,xmm3
19 436 mov eax,ebx
20 438 jmp 371 (27062973)
```

Figure 4. JIT compiler generated code for the `for` loop in the `average` function

proposal is under active development and changes are likely to happen as the proposal is being refined and moves forward through the approval process.

As an example use case, Figure 3 shows the SIMD JavaScript code for computing the average of an array of floating point numbers. The numbers are held in a Float32Array typed array; `data`. The benefit of using SIMD operations, for computing the average, is that four numbers can be added in one operation, thereby reducing the number of iterations by a factor of 4, and achieving an equivalent speedup.

The optimizing Just-In-Time (JIT) compiler in our Chrome/V8 SIMD enabled prototype is able to produce the code in Figure 4 for the body of the loop. The code shows how the compiler is able to utilize 128-bit SIMD registers (xmm) to hold the value of `sum` and to use the `addps` instruction for adding 4 single precision numbers in one instruction. The details for this code snippet are as follows:

line 1-2: Check the loop index, `i`, against the upper bound and exit the loop if upper bound is encountered. `eax` holds the loop index and `edx` holds the upper bound.

line 3-4: Check to enable the JavaScript engine to abort the execution, if the loop has been running for too long. It will prevent a user program from hanging the browser.

line 5-11: Bounds check for the SIMD_float32x4_load operation. `eax` holds the index and `ecx` holds the upper bound.

line 12: Load the 4 32-bit float values. `edi` holds the base address of the data. The reason the index is multiplied by 2 and not 4 is the fact that V8 represents integers in bits 1-31, so the value in `eax` is already holding the value of the index variable times 2.

line 13-14: Add the four 32-bit float values.

```
float averageScalar(float *a, uint32_t length) {
  float sum = 0.0f;
  for (uint32_t j = 0, l = length; j < l; j = j + 4) {
    sum = sum + (*(a++));
  }
  return sum/length;
}
```

Figure 5. Scalar C code for the `average` function

line 15-17: increment the loop index by 4. Since the integer representation used by V8 is done in bits 1-31 the actual value added is 8. The overflow checks is there to ensure that the result can still be represented in 31 bits, if not the representation is switched to a boxed floating point number.

line 18: Move the result of the SIMD add operation, to the xmm1 register, which holds the value of `sum`.

line 19-20: Move the new loop index value to it's proper register and jump back to the top of the loop.

For more details on how the JIT compilers operate see [9].

3.1 The Future of SIMD.JS

The proposal has been presented to TC39, the JavaScript language standard commitee, and was unanimously approved for stage 1 in 2014. Stage 1 is the proposal stage. It indicates that the need has been justified, and an outline for a solution has been accepted. It does not mean that this is the final proposal.

The focus, so far, has been on identifying types and operations that can be effectively implemented on all relevant CPU architectures. We realize that CPUs have destinct features that are useful and it will make sense to expose such features to the JavaScript programmer. This will most likely be done via architecture specific extensions to the SIMD object, e.g. SIMD.x86.*

SIMD.JS is currently being refined and prepared for the next stages of approval, and we expect this to be part of the EcmaScript 7 standard (ES7). EcmaScript 5 is the current JavaScript standard. EcmaScript 6 is slated for a mid-2015 release. ES6 is a major overhaul of the JavaScript language and a substantial set of new features were added, as reflected by the size of the language specification document. The ES5 specification document is roughly 300 pages, whereas the ES6 specification is roughly double that. Most browsers have already implemented most of the ES6 features.

4. Emscripten

Emscripten is a compiler that compiles C/C++ programs into JavaScript. It is based on the clang/LLVM compiler infrastructure [3]. The compiler is the brainchild of Mozilla's Alon Zakai.

As an example of how it works, we'll look at the generated JavaScript code resulting from compiling a simple C function. We'll again use a function that computes the average of a sequence of float numbers. Figure 5 shows the input C program.

The Emscripten compiler command is similar to the clang compiler command, and takes most of the same options. The following command will generate the optimized JavaScript code shown in Figure 6:

```
$ emcc -O2 -g average-scalar.c
```

This example shows how Emscripten manages to map a staticly typed language with pointers (C) to a dynamically typed language without pointers (JavaScript).

Memory is modelled as overlayed typed arrays. In this example when the pointer `*a` is used to fetch from memory the corresponding JavaScript code is `+HEAPF32[$a$addr$06 >> 2]` (line 14). `HEAPF32` is a global JavaScript typed array declared as follows:

```
1  function _averageScalar($a, $length) {
2    $a = $a | 0;
3    $length = $length | 0;
4    var $a$addr$06 = 0, $add = 0.0, $j$05 = 0,
5        $sum$0$lcssa = 0.0, $sum$04 = 0.0, sp = 0;
6    sp = STACKTOP;
7    if (($length | 0) == 0)
8      $sum$0$lcssa = 0.0;
9    else {
10     $a$addr$06 = $a;
11     $j$05 = 0;
12     $sum$04 = 0.0;
13     while (1) {
14       $add = $sum$04 + +HEAPF32[$a$addr$06 >> 2];
15       $j$05 = $j$05 + 4 | 0;
16       if (($j$05 >>> 0 < $length >>> 0)) {
17         $sum$0$lcssa = $add;
18         break;
19       } else {
20        $a$addr$06 = $a$addr$06 + 4 | 0;
21        $sum$04 = $add;
22       }
23     }
24   }
25   STACKTOP = sp;
26   return +($sum$0$lcssa / +($length >>> 0));
27 }
```

Figure 6. JavaScript code generated by Emscripten for the `averageScalar` function

```
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);
```

All of these typed arrays are views on the same array buffer, so they all access the same physical memory. Notice that the index expression `'$a$addr$06 >> 2'` is shifted right by 2. This is because `$a$addr$06` is a byte address, and elements in the `HEAPF32` are 4 bytes each.

To enable the JavaScript JIT compilers to generate efficient code two type coercision tricks are used.

For integers and pointers the `'expr | 0'` is used to guarantee that the type of the resulting expression is a 32-bit integer. JavaScript semantics of the the bitwise | expression dictate that the resulting expression is a 32-bit integer. A side effect of pointers being 32-bit integers is that compiled C/C++ programs are restricted to a 32-bit address space.

For floating point numbers, the unary '+' operator is applied, because JavaScript semantics dictate that the resulting expression is a floating point number.

Emscripten has been successfully used to compiler very large C/C++ code bases (+100K lines of code). For example both Epic's and Unity's game engines have been ported, using Emscripten [4]. Game engines are one example of software that will have optional implementations of performance critical portions of the code implemented using SIMD features. Since, JavaScript hasn't had a way of utilizing these powerful low level SIMD features of the CPU, Emscripten has not been able to compile these highly tuned implementations of the performance critical sections of the code. However, with the introduction of SIMD.JS, Emscripten will

```
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++)));
  }
  float mSumx4[4];
  _mm_storeu_ps(mSumx4, sumx4);
  return (mSumx4[0] + mSumx4[1] +
          mSumx4[2] + mSumx4[3])/length;
}
```

Figure 7. SIMD C code with intrinsics for the `average` function

```
1  function _averageIntrin($a, $length) {
2    $a = $a | 0;
3    $length = $length | 0;
4    var $add$i = SIMD_float32x4(0, 0, 0, 0),
5        $j$09 = 0,
6        $sumx4$0$lcssa = SIMD_float32x4(0, 0, 0, 0),
7        $sumx4$010 = SIMD_float32x4(0, 0, 0, 0),
8        sp = 0;
9    sp = STACKTOP;
10   if (($length | 0) == 0)
11     $sumx4$0$lcssa = SIMD_float32x4_splat(Math_fround(0));
12   else {
13     $j$09 = 0;
14     $sumx4$010 = SIMD_float32x4_splat(Math_fround(0));
15     while (1) {
16        $add$i =
17          SIMD_float32x4_add(
18            $sumx4$010,
19            SIMD_float32x4_load(
20            buffer, $a + ($j$09 << 2) | 0));
21        $j$09 = $j$09 + 4 | 0;
22        if (($j$09 >>> 0 < $length >>> 0)) {
23          $sumx4$0$lcssa = $add$i;
24          break;
25        } else $sumx4$010 = $add$i;
26     }
27   }
28   STACKTOP = sp;
29   return +((+$sumx4$0$lcssa.w + (+$sumx4$0$lcssa.z +
30          (+$sumx4$0$lcssa.x + +$sumx4$0$lcssa.y))) /
31          +($length >>> 0));
32 }
```

Figure 8. Emscripten generated SIMD JavaScript code for the `averageIntrin` function

now be able to take full advantage of those. The next section covers how this is accomplished.

5. Compiling C++ with SIMD intrinsics

Figure 7 shows a typical SIMD implementation of the `average` function in C, using x86 SIMD intrinsics.

The `__m128` type holds 4 32-bit float numbers. The `_mm_*_ps` function calls are the SIMD intrinsics, which operates on single precision `__m128` values. For example, the `_mm_add_ps` intrinsic maps to the x86 `addps` instruction, which adds 4 32-bit float numbers in one operation. This allows the iteration count to be reduced by a factor of 4, resulting in an equivalent speedup.

The resulting JavaScript produced by Emscripten is shown in Figure 8.

The JavaScript code in Figure 8 might appear extensive at first look, however, it is very similar to the handwritten version of the function from Figure 3. The while loop starting at line 15 corresponds to the for loop from the C program. Implementing a for loop as a while loop takes a bit more code. The important

```
typedef float floatx4 __attribute__ ((vector_size(16)));
float averageVectorSize(float *a, uint32_t length) {
  floatx4 sumx4 = {0.0f, 0.0f, 0.0f, 0.0f};
  floatx4 *ax4 = (floatx4 *)a;
  for (uint32_t j = 0, l = length; j < l; j = j + 4) {
    sumx4 = sumx4 + *(ax4++);
  }
  return (sumx4[0] + sumx4[1] +
          sumx4[2] + sumx4[3])/length;
}
```

Figure 9. SIMD C code with `vector_size` for the `average` function

thing to notice here is the use of the `SIMD_float32x4_add` and `SIMD_float32x4_load` SIMD.JS operations. The use of '_' instead of '.' is because Emscripten have created single identifier versions for all the SIMD.JS primitives.

Use of CPU specific intrinsics makes the C version of the code target specific, i.e. it will only execute on x86 CPUs, whereas the resulting JavaScript code will execute on all architectures supported by the underlying SIMD enabled JavaScript engine.

Use of non target specific SIMD code in C is possible via the gcc `vector_size` attribute. Emscripten also supports compiling such code. A non target specific version of the `average` function is shown in Figure 9. The generated JavaScript code is virtually identical to the code resulting from the C code using intrinsics. If possible, developers should be encouraged to write their SIMD code using this more universal syntax.

6. Benchmarks

We've created a set of benchmark kernels. The kernels are written in both C++ and JavaScript. Each kernel will have both a scalar implementation and a SIMD implementation.

The C++ SIMD implementation is done using x86 intrinsics. The C++ kernels are compiled with both the C++ clang compiler and with the Emscripten compiler. This gives us a basis for comparing SIMD/Scalar speedups for both C++ and JavaScript as well as C++/JavaScript performance differences. For the JavaScript execution we've used our two SIMD enabled JavaScript engines; V8 and SpiderMonkey.

The benchmarks are written such that each kernel operation is executed as many times as it takes for it to run in about a second. This guarantees that the optimizing JavaScript JIT compilers kick in, which is essential for optimimal performance.

When creating SIMD optimized code from scalar code, there are basically two approaches; 1) Combine a sequential sequence of scalar operations into SIMD operations, and 2) Combine multiple iterations of simple loops, by replacing the scalar operations with equivalent SIMD operations. Typical examples of 1) are matrix/vector operations, where there's no loops involved with the computations, and a typical example of 2) is the average example shown previously. We cover both of these two types of SIMD optimizations in the benchmarks we've written.

The benchmark kernels we've collected performance data for are:

- **AverageFloat32x4:** Average 10,000 32-bit float number. The SIMD version of this kernel will use SIMD instructions in the loop to perform vector versions of the equivalent scalar operations, thereby achieving an iteration count reduced by the vector length (4). This the kernel for the example used throughout the previous examples.
- **Mandelbrot:** [1] Compute how many iterations it takes for z(i+1) = z(i)**2 + c to diverge for seed point c = (0.01, 0.01).

Divergence is determined by z**2 >4. Different seed points will typically diverge after different number of iterations. This results in a non-trivial SIMD/vector implementation, since the control flow in the loop will differ for different seed points. Our implementation relies on computing an increment vector, with 0 for the vector elements that have already diverged, and 1 for the elements that haven't. However, for the chosen seed point the series never diverges, so the loop always runs up to the maximum number of iterations allowed. The maximum number of iterations for this kernel is set to 100. The scalar version of the kernel will compute the iteration count for one seed point, and the SIMD version will compute the iteration count for four seedpoints. As an example of how all the benchmark kernels are structured we've shown the handwritten JavaScript and C++ version of this kernel in Figure 11 and Figure 10.

- **MatrixMultiplication:** Multiply two 4x4 matrices. For each element in the resulting 4x4 matrix, 4 scalar multiplications and 3 scalar additions are required. The SIMD version will rearrange the source data, using a shuffle operation, and compute an entire row in the result matrix with 4 SIMD multiplies and 3 SIMD additions.
- **VertexTransform:** Multiply a 4x4 matrix with a 4 element vector. This is a common CPU side operation for creating projection matrices for webGL shaders. Typically, it's used to compute a transformation for a point in 3D space, e.g. rotation around an axis. For each element in the resulting 4 element vector, 4 scalar multiplies and 3 scalar additions must be computed. The SIMD version will compute all 4 elements, using SIMD multiplies and adds, thereby reducing the number of multiply and add instructions. Some shuffling is required to get the input data into the right lanes.
- **MatrixTranspose:** Transpose a 4x4 matrix. This is also a common operation, when doing vector/matrix algebra. Rows are made into columns. The scalar kernel, simply moves the 16 elements around one by one. The SIMD version uses 8 shuffle operations.
- **MatrixInverse:** Compute the inverse of a 4x4 matrix. This is a complex operation, involving hundreds of multiples and add operations. There are several different ways of computing the inverse of a matrix. Here we have chosen a method called the Cramer rule [5]. This kernel is the most compute intensive.

The sources for the C++ benchmarks can be found here [6], and the sources for the handwritten JavaScript benchmarks can be found here: [7]

7. Results

We've collected performance results for these combinations:

- Natively compiled C++
- Handwritten JavaScript executed with V8
- Emscripten produced JavaScript from C++ implementation executed with V8
- Emscripten produced JavaScript from C++ implementation executed with SpiderMonkey

All performance numbers were collected while running on an Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz.

Note, we're not yet able to run 'generic'/handwritten SIMD JavaScript code efficiently with SpiderMonkey. SpiderMonkey uses two different JIT compilers; one for 'generic' JavaScript code (IonMonkey) and another for the asm.js JavaScript subset (Odin-

```
__m128i mandelx4(__m128 c_re4, __m128 c_im4,
                 uint32_t max_iterations) {
  __m128 z_re4 = c_re4;
  __m128 z_im4 = c_im4;
  __m128 four4 = _mm_set_ps1(4.0f);
  __m128 two4 = _mm_set_ps1(2.0f);
  __m128i count4 = _mm_set1_epi32(0);
  __m128i one4 = _mm_set1_epi32(1);
  uint32_t i;
  for (i = 0; i < max_iterations; ++i) {
    __m128 z_re24 = _mm_mul_ps(z_re4, z_re4);
    __m128 z_im24 = _mm_mul_ps(z_im4, z_im4);
    __m128 mi4 =
      _mm_cmple_ps(_mm_add_ps(z_re24, z_im24), four4);
    if (_mm_movemask_ps(mi4) == 0)
      break;
    __m128 new_re4 = _mm_sub_ps(z_re24, z_im24);
    __m128 new_im4 = _mm_mul_ps(_mm_mul_ps(two4, z_re4), z_im4);
    z_re4 = _mm_add_ps(c_re4, new_re4);
    z_im4 = _mm_add_ps(c_im4, new_im4);
    count4 = _mm_add_epi32(
      count4, _mm_and_si128(_mm_castps_si128(mi4), one4));
  }
  return count4;
};
```

Figure 10. C++ SIMD Mandelbrot kernel

```
function mandelx4(c_re4, c_im4, max_iterations) {
  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);
    // check if all 4 values are greater than 4.0
    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;
}
```

Figure 11. JavaScript SIMD Mandelbrot kernel

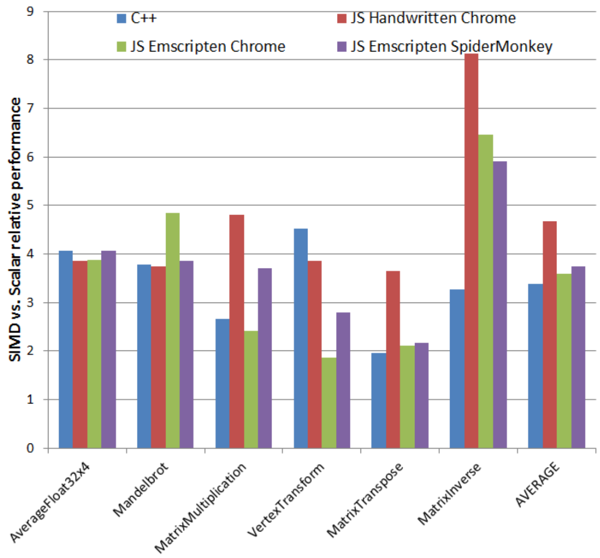

Figure 12. SIMD vs. Scalar relative speedups

Monkey). Only the OdinMonkey JIT compiler has been adapted to work with the SIMD operations.

7.1 SIMD vs. Scalar

Figure 12 shows the relative SIMD vs. Scalar performance of the four combinations. Greater than 1 means that the kernel ran that the SIMD kernel ran that much faster than the corresponding scalar kernel.

7.2 Scalar C++ vs. JavaScript

Figure 13 shows the relative scalar C++ vs. JavaScript performance for each of the four combinations. Less than 1 means that the JavaScript kernel ran that much slower than the corresponding C++ kernel.

7.3 SIMD C++ vs. JavaScript

Figure 14 shows the relative SIMD C++ vs. JavaScript performance for each of the four combinations. Less than 1 means that the JavaScript kernel ran that much slower than the corresponding C++ kernel.

8. Future Work

For future work we're planning on adding a more complete set of SIMD.JS primitives, to better cover all the existing SIMD intrinsics being used. A study of which missing intrinsics have wider use is warranted. When adding more primitives, with direct mapping to hardware instructions on one architecture, but not on another, it might not be possible to achieve the desired performance across all architectures.

9. Summary

This paper describes SIMD.JS and Emscripten, together these technologies allow porting of compute intensive native C/C++ applications to the Web platform with near-native performance. The contributions of our work include the changes to Emscripten enabling compilation of C/C++ programs that use SIMD intrinsics or gcc style vector code. Our contributions are publicly available, and our evaluation demonstrates that the SIMD vs. scalar speedup we see

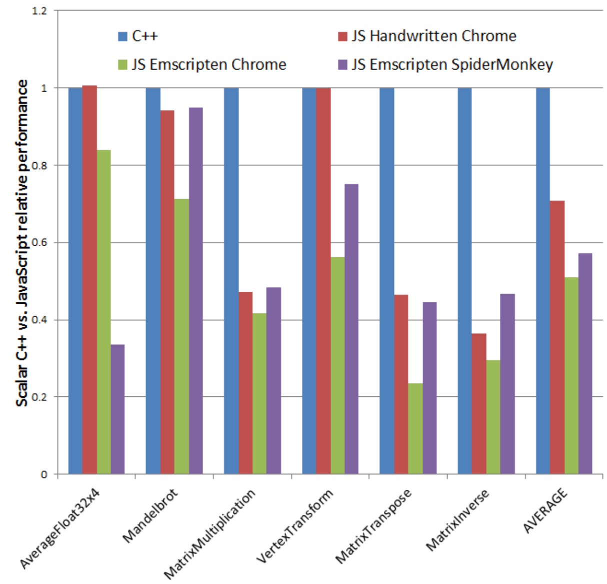

Figure 13. Scalar C++ vs. Javascript relative performance

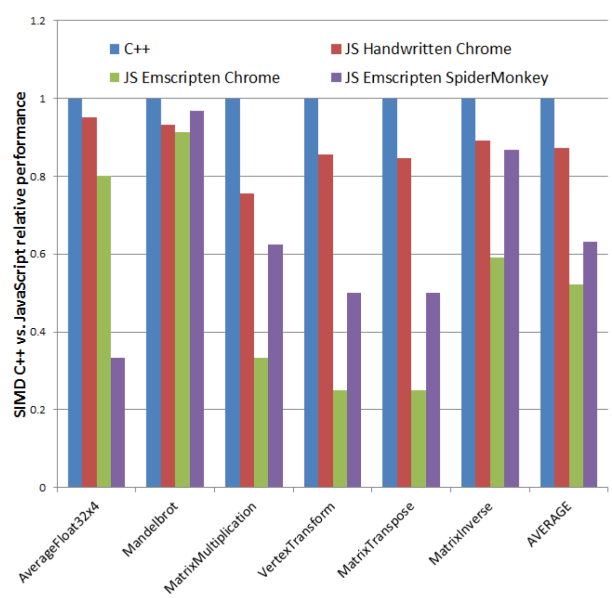

Figure 14. SIMD C++ vs. Javascript relative performance

with the C++ kernels are equivalent for the equivalent JavaScript kernels.

Acknowledgments

Acknowledgments goes here.

References

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

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

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

[4] Unreal Engine: Getting Started: Developing HTML5 Projects, February 2014. https://docs.unrealengine.com/latest/INT/Platforms/HTML5/GettingStarted/index.html.

[5] Cramer's rule, February 2015. http://en.wikipedia.org/wiki/Cramer%27s_rule.

[6] SIMD C++ Benchmarks, February 2015. https://github.com/PeterJensen/benchcpp.

[7] SIMD JavaScript Benchmarks, February 2015. https://github.com/johnmccutchan/ecmascript_simd/tree/master/src/benchmarks.

[8] 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.

[9] John McCutchan, Haitao Feng, Nicholas D. Matsakis, Zachary Anderson, and Peter Jensen. A SIMD Programming Model for Dart, JavaScript, and other dynamically typed scripting languages, February 2014. https://e9fe7ff0-a-62cb3a1a-s-sites.googlegroups.com/site/wpmvp2014/paper_18.pdf.