SIMD in JavaScript via C++ and Emscripten

Peter Jensen
Intel Corporation
peter.jensen@intel.com

John McCutchan
Google Inc.
johnmccutchan@google.com

Ivan Jibaja
Intel Corporation
ivan.jibaja@intel.com

Dan Gohman
Mozilla
sunfish@mozilla.com

Ningxin Hu
Intel Corporation
ningxin.hu@intel.com

Abstract

This is the text of the abstract.

1. Introduction

We'll explore the use of Mozilla's Emscripten to compile C++ programs, that has use of SIMD intrinsics or gcc style vector code, into JavaScript. This was recently made possible by the SIMD.JS primitives introduced in JavaScript engine prototypes for Chromium and Firefox as well as extensions to the Emscripten compiler. Emscripten will correctly translate a subset of available C++ SIMD x86 intrinsics into corresponding operations defined in SIMD.JS. The JavaScript benchmarks associated with the SIMD.JS primitives were converted to C++ by hand, and then automatically converted back into JavaScript using the Emscripten compiler.

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

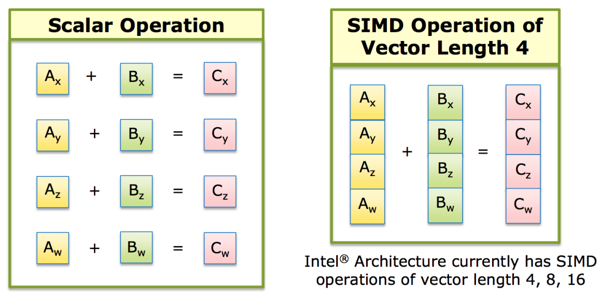

Figure 1. Replacing four scalar additions with one SIMD addition

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 use 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.

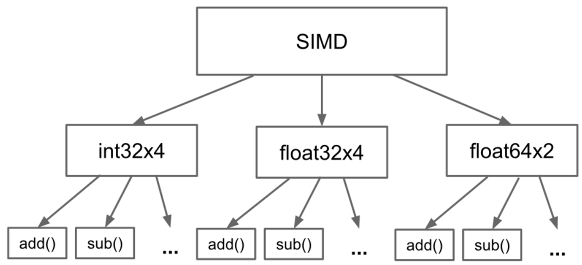

Figure 2. SIMD.JS object hierarchy

```
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

```
Assembly dump
```

Figure 4. JIT compiler generated code for the average function

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 ¡REF¿. 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 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 utilize a 128-bit SIMD register (xmm) to hold the value of `sum` and to use the `addps` instruction for adding 4 single precision numbers in one instruction. For more details on how the JIT compilers operate see ¡REF¿.

2.1 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.

```
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

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.

3. Emscripten

Emscripten is a compiler that compiles C/C++ programs into JavaScript. It is based on the clang/LLVM compiler infrastructure ¡REF¿. The compiler is the brainchild of Alon Zakai of Mozilla.

As an example of how it works, we'll look at the generated JavaScript code resulting from compilign a simple C function. We'll again use a function that computes the average 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 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 (C) with pointers 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:

```
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.

```
function _averageScalar($a, $length) {
  $a = $a | 0;
  $length = $length | 0;
  var $a$addr$06 = 0, $add = 0.0, $j$05 = 0,
      $sum$0$lcssa = 0.0, $sum$04 = 0.0, sp = 0;
  sp = STACKTOP;
  if (($length | 0) == 0)
    $sum$0$lcssa = 0.0;
  else {
    $a$addr$06 = $a;
    $j$05 = 0;
    $sum$04 = 0.0;
    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 = $add;
      }
    }
  }
  STACKTOP = sp;
  return +($sum$0$lcssa / +($length >>> 0));
}
```

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

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

4. 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 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.

It's important to note that 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.

```
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(buffer, $a + ($j$09 << 2) | 0));
20        $j$09 = $j$09 + 4 | 0;
21        if (($j$09 >>> 0 < $length >>> 0)) {
22          $sumx4$0$lcssa = $add$i;
23          break;
24        } else $sumx4$010 = $add$i;
25     }
26   }
27   STACKTOP = sp;
28   return +((+$sumx4$0$lcssa.w + (+$sumx4$0$lcssa.z +
29         (+$sumx4$0$lcssa.x + +$sumx4$0$lcssa.y))) /
30         +($length >>> 0));
31 }
```

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

```
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

5. Benchmarks

6. Results

7. Summary

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...