



NASoftware Limited
Incorporating InfoSAR

Translation of PowerPC/AltiVec SIMD Macros to IA32/SSE/AVX

Peter Cromwell, Ian McConnell*
N.A. Software Ltd

This work was funded by Intel Corporation with whom copyright exists.

Revision 2.0; Released 25th September 2010

Contents

1	Introduction	2
2	Requirements and use	3
2.1	Compiler dependency	3
3	Endian issues	3
4	Efficiency issues	4
5	Accuracy and rounding	5
6	Instructions that are simulated	5
6.1	Instructions with SIMD simulations	5
6.2	Instructions with serial simulations	6
7	Instructions that are ignored	6
8	Instructions added at release 2	6
9	Benchmarks	7
9.1	Vector Add	7
9.2	Vector Sine	8
9.3	Fourier Transform	10
9.4	Rounding	10
10	AVX Considerations	12
10.1	AVX Features	12
10.2	AVX support in GCC	12
10.3	Register Pressure: Help from GCC	13
10.3.1	4×4 Matrix Multiply	14
10.3.2	Vector Sine	14
10.4	Benchmarks on AVX machine	15
10.4.1	Inner product	15
10.4.2	4×4 matrix transpose	16
10.4.3	Multiply two 4×4 matrices	16
A	4x4 matrix multiply	18
B	Inner product	19

1 Introduction

This document is aimed at experienced developers who need to migrate their existing vector-oriented C/C++ PowerPC AltiVec code to the Intel x86 (Intel Architecture 32-bit) SSE (Streaming SIMD Extensions) extensions.

The SIMD facilities of the PowerPC/AltiVec chip can be accessed from C code using the `altivec.h` header file. This makes the interface defined in the Motorola specification *AltiVec Technology Programming Interface Manual* available through a set of macros that target the SIMD assembly instructions.

This document accompanies a modified version of the `altivec.h` file (`altivec2avx.h`) which uses the same Motorola interface but targets Intel processors with the SSE2 level of SIMD support.

This can be done because, at the conceptual level, AltiVec and SSE are quite similar. They are single instruction/multiple data (SIMD) vector units with 4×32 bit vectors that prefer to be 16 byte aligned. The vectors are accessed through a C programming interface which treats them as a special 128 bit data type with a set of function-like intrinsics. Intel used the `_mm_` prefix whereas AltiVec has `vec_`. Generally, there is a good correlation between the two instruction sets with about two thirds of the most commonly used functions directly translatable. For the remaining AltiVec instructions, many can be emulated with a few SSE calls.

Places that are likely to require a programmer's attention include:

- code for handling misaligned data
- use of instructions that saturate the result on overflow
- use of instructions in which the position of individual elements within a SIMD vector matters (*e.g.* `pack`, `perm`).

In the first case, reading misaligned data is much simpler in SSE — you no longer need to load two adjacent aligned vectors and extract the required elements, you just load an unaligned vector with `_mm_loadu_si128` or `_mm_loadu_ps` for integer or float data. The last case is a problem caused by the change in endian format (see §3).

Note that the kind of translation provided (direct mapping, inline function, none) may vary with the data type. If the instruction you want is not translated, check to see whether switching between signed and unsigned, or changing the width of an integer type will help.

For more information on porting code between AltiVec and SSE see:

AltiVec/SSE Migration Guide,

http://developer.apple.com/documentation/performance/Conceptual/Accelerate_sse_migration/

SSE Performance Programming,

<http://developer.apple.com/hardwaredrivers/ve/sse.html>

2 Requirements and use

You will need to have the standard header files `xmmintrin.h` and `emmintrin.h` installed on your system to provide access to the SSE and SSE2 instructions.

Replace the original `altivec.h` header file you use for compiling on a PowerPC with the file `altivec2avx.h` containing SSE translation macros.

Recompile your code on an Intel machine using the appropriate flag to enable the SSE extensions, typically `-msse2`. You may also need the flag `-flax-vector-conversions` to remove superfluous type conversion errors.

2.1 Compiler dependency

In the Motorola specification, the same high-level function name is used with multiple data types and the compiler selects the correct instruction based on the types of the arguments the function is called with. The `altivec2avx.h` header file has been derived from one that was supplied with `gcc`. It makes use of some internal compiler macros to fake the function overloading in the C interface. This is probably not portable between compilers so you may need access to `gcc`. Fortunately, the compilers supplied for use on real-time systems or for building DSP applications are often based on `gcc`.

3 Endian issues

AltiVec is big-endian and SSE is little-endian. Both units represent their data with IEEE-754 floating point format, but the Intel architecture stores the results in little-endian order. So, for the array

```
float data[4]= { 10.f, 20.f, 30.f, 40.f };  
vector float v = _mm_loadu_ps(&data[0]);
```

the data in `v` will look like this:

```
v = {40.f, 30.f, 20.f, 10.f}
```

On the PowerPC values stored in memory and values stored in a processor register are represented in the same way. On Intel processors, the order of the bytes is reversed when data are transferred between registers and memory.

If your code is pure SIMD so that you always apply identical operations to all elements of a vector then this will not affect you. When the results are saved to memory, the Intel architecture restores the byte order to what would be expected.

If you refer to left/right, high/low or even/odd locations within a vector and you are getting incorrect results, this is very likely to be the source of the problem.

In the case of a vector register, the entire vector is byte swapped: not only are the bytes representing each element reversed, but also the order of the elements within a vector. The

first is essentially invisible to the programmer but the second affects the indexing in the vector. Any instruction that addresses a particular element in a vector will access the wrong location. For example, it interchanges left and right shifts when permuting elements.

In general, a macro does have sufficient context to determine whether the elements of a vector register should be indexed in register order or memory order. The following instructions have been implemented with both register order and memory order versions. By default you will get the register order implementation, which is a literal translation of the AltiVec macro. If you define the macro `MEMORY_ORDER` before reading in `altivec2avx.h` you will get the memory order implementation, which reverses the indexing order.

- splat: `vec_splat`
- merge: `vec_merge`, `vec_mergel`
- shift octet: `vec_slo`, `vec_sro`
- multiply even/odd: `vec_mule`, `vec_mulo`
- unpack: `vec_unpackh`, `vec_unpackl`.

Fortunately, one of the main uses of the AltiVec permute unit is in loading and storing misaligned data but, as mentioned above, SSE provides easier ways to do this.

4 Efficiency issues

Do not expect your highly tuned AltiVec code to be translated into high-performance SSE. You will get a quick and easy first cut.

In AltiVec you can gain a speed advantage by unrolling up to eight-way in parallel. On x86, there are fewer registers and unrolling may overflow the register file causing a large number of extra loads. In this case returning to the simpler, unrolled code may give a performance advantage.

If you want high-performance SSE, you may need to re-examine your algorithm to account for the out-of-order execution and fewer registers. Some of the issues are discussed below.

Whether an AltiVec instruction maps onto a single SIMD instruction, a sequence of SIMD instructions, or a serial implementation can depend on the data type. You may be able to switch to a faster instruction by changing the width or signed/unsigned condition of an integer.

AltiVec logical shift and arithmetic shift instructions operate independently on each element of a SIMD vector, but the SSE equivalents shift all elements by the same amount. The AltiVec behaviour is simulated with serial implementations. If you do not need this generality, you will get better performance by changing your code to call the SSE functions.

5 Accuracy and rounding

Any algorithm using floating point calculations being ported from the AltiVec to SSE should be tested for numerical accuracy. While both processors store their data with IEEE-754 floating point format, the AltiVec design is based upon a fused multiply add. The Intel processor separates these into a multiply and an add operation and so may incur an extra rounding step.

The Intel processor does, however, support more rounding modes (*nearest*, *zero*, *Inf* and *-Inf*) compared to the *round to nearest* of the AltiVec.

6 Instructions that are simulated

Each of the following AltiVec instructions has no direct SSE equivalent but its effect has been produced by combining sequences of SIMD instructions, or with a serial implementation that processes each element of the vector in turn. Whether simulation is necessary may depend on the data type: some AltiVec instructions have direct translations for some but not all of `signed/unsigned`, `char`, `short`, and `int`.

6.1 Instructions with SIMD simulations

- absolute value: `vec_abs`, `vec_abss`
- average: `vavguw`, `vavgsh`, `vavgsw`
- rounding: `vrfiz`, `vrfig`, `vrfig`, `vrfin`
- type conversion: `vcfsx`, `vctsx`
- unsigned comparison: `vcmpgtub`, `vcmpgtuh`, `vcmpgtuw`
- bounds checking: `vcmpbfp`
- logical nor: `vnor`
- maximum: `vmaxsb`, `vmaxuh`, `vmaxuw`, `vmaxsw`, `vmaxfp`
- minimum: `vminsb`, `vminuh`, `vminuw`, `vminsw`, `vminfp`
- select: `vec_sel`
- shift octet: `vslo`, `vsro`
- splat: `vspltb`, `vsplth`, `vspltw`, `vspltf`.
- approximation: `vec_expte`, `vec_loge`
- unsigned type conversion: `vec_vcfux`, `vec_vctux`

Cache hints are also ignored. The LRU variants `vec_ldl` and `vec_stl` of the load and store instructions that mark cache lines as ‘least recently used’ are `#defined` to the ordinary `vec_ld` and `vec_st`.

6.2 Instructions with serial simulations

- load/store by element: `vec_lde`, `vec_ste`
- multiply even/odd: `vec_mule`, `vec_mulo`
- multiply-sum: `vec_msum`
- pack/unpack: `vec_pack`, `vupkhsb`, `vupkhsh`, `vupklhsb`, `vupklsh`
- logical shift: `vslb`, `vslh`, `vslw`, `vsrb`, `vsrh`, `vsrw`
- arithmetic shift: `vsrab`, `vsrah`, `vsraw`
- rotate: `vrlb`, `vrlh`, `vrlw`
- long shift: `vec_sll`, `vec_srl`.

7 Instructions that are ignored

The following AltiVec instructions are `#defined` to be empty macros, or zero if they return a value. You will not get an error by using them — they will just be ignored.

- cache touches: `vec_dss`, `vec_dssall`, `vec_dst`, `vec_dstst`, `vec_dstt`, `vec_dststt`
- status register: `vec_mfvscr`, `vec_mtvscr`.

8 Instructions added at release 2

The following AltiVec instructions do not have direct SSE equivalents and were not implemented at Release 1 — they gave an ‘undefined reference’ error on linking code that uses them. For convenience, they have now all been given C implementations.

For most of these instructions the position of individual elements within the vector matters (see §3) or the AltiVec instructions saturate the result on overflow. In the first case a programmer should analyse the code to ensure the correct order of elements is used; in the second case it may be possible to substitute the non-saturated version of an instruction at a modest efficiency gain.

- include carry: `vec_addc`, `vec_subc`
- saturate result: `vec_vaddsws`, `vec_vadduws`, `vec_madds`, `vec_mradds`, `vec_msums`, `vec_vsubsws`, `vec_vsubuws`
- saturated pack: `vec_vpkshus`, `vec_vpkswus`, `vec_vpkuwus`

- saturated sum across vector: `vec_sums`, `vec_sum2s`, `vec_sum4s`
- pack/unpack pixels: `vec_packpx`, `vec_vupkhp`, `vec_vupklp`
- multiply low and add: `vec_mladd`
- create shift vectors for unaligned data: `vec_lvsl`, `vec_lvslr`
- permutation: `vec_perm`,
- double shift: `vec_sld`.

The macros for `vec_ld` and `vec_st` were updated to allow compiling `altivec2avx.h` on a 64 bit machine.

9 Benchmarks

We give here examples of PowerPC/Altivec vector codes as run on PowerPC and then on Intel using the `altivec2avx.h` include file. The systems used to provide timings are:

Linux/Intel: an Intel Core 2 (T7200), 2GHz, 4Mb cache.

Linux/PPC: a MPC8641HPCN board (7448), 1.5GHz.

These have different clock speeds; we factor this out by giving the number of clock cycles used per vector element.

9.1 Vector Add

The Altivec code to sum two arrays of floating point vectors

```
float *block_out, *block_in1, *block_in2;
for (i = 0; i < v_length; i += 4) {
    vector float Avf32, Bvf32, Rvf32;
    Avf32 = vec_ld(0, &block_in1[i]);
    Bvf32 = vec_ld(0, &block_in2[i]);
    Rvf32 = vec_add(Avf32, Bvf32);
    vec_st(Rvf32, 0, &block_out[i]);
}
```

will run on SSE with the inclusion of `altivec2avx.h`.

```
#define MEMORY_ORDER
#include "altivec2avx.h"
```

Timings in instruction cycles per array element are given below.

Vector length	Linux/Intel	Linux/PPC
000256	1.1	1.5
001024	1.2	1.5
004096	1.8	3.4
016384	1.8	3.4
065536	1.9	6.0
131072	1.8	21.3

Timings in μ seconds:

Vector length	Linux/Intel	Linux/PPC
000256	0.1	0.3
001024	0.6	1.0
004096	3.6	9.2
016384	14.7	36.7
065536	61.8	261.2
131072	120.2	1859.0

9.2 Vector Sine

Model Altivec code to calculate the sine of an array of floating point numbers while taking account of the vector length of four might look like:

```
float *block_out, *block_in;
for (i = 0; i < v_length; i += 4) {
    vector float Avf32, Rvf32;
    Avf32 = vec_ld(0, &block_in[i]);
    Rvf32 = vsin(Avf32);
    vec_st(Rvf32, 0, &block_out[i]);
}
```

A fast vector sine algorithm is given in "*A Fast, Vectorizable Algorithm for Producing Single-Precision Sine-Cosine Pairs*" available from <http://arxiv.org/pdf/cs.MS/0406049>

The code is reproduced below

```
vector float vsin(vector float v)
{
    vector float s1, s2, c1, c2, fixmag1;
    vector float vzero = VEC_CONST(0.0);
    vector float vone = VEC_CONST(1.0);
    vector float vtwo = VEC_CONST(2.0);
    vector float vhalfpi = VEC_CONST(1.0/(2.0*3.1415926536));
    vector float v_ss1 = VEC_CONST( 1.5707963268);
    vector float v_ss2 = VEC_CONST(-0.6466386396);
    vector float v_ss3 = VEC_CONST( 0.0679105987);
    vector float v_ss4 = VEC_CONST(-0.0011573807);
    vector float v_cc1 = VEC_CONST(-1.2341299769);
```

```

vector float v_cc2 = VEC_CONST( 0.2465220241);
vector float v_cc3 = VEC_CONST(-0.0123926179);

vector float x1 = vec_madd(v, vhalfpi, vzero);
/* q1=x/2pi reduced onto (-0.5,0.5), q2=q1**2 */
vector float q1 = vec_nmsub(vec_round(x1), vone, x1);
vector float q2 = vec_madd(q1, q1, vzero);
s1= vec_madd(q1,
            vec_madd(q2,
                    vec_madd(q2, vec_madd(q2, v_ss4, v_ss3), v_ss2),
                    v_ss1),
            vzero);
c1= vec_madd(q2,
            vec_madd(q2, vec_madd(q2, v_cc3, v_cc2), v_cc1),
            vone);
/* now, do one out of two angle-doublings to get sin & cos theta/2 */
c2 = vec_nmsub(s1, s1, vec_madd(c1, c1, vzero));
s2 = vec_madd(vtwo, vec_madd(s1, c1, vzero), vzero);
/* now, cheat on the correction for magnitude drift...
   if the pair has drifted to (1+e)*(cos, sin),
   the next iteration will be (1+e)**2*(cos, sin)
   which is, for small e, (1+2e)*(cos,sin).
   However, on the (1+e) error iteration,
   sin**2+cos**2=(1+e)**2=1+2e also,
   so the error in the square of this term
   will be exactly the error in the magnitude of the next term.
   Then, multiply final result by (1-e) to correct */
/* must use this method with un-normalized series, since magnitude error is large */
fixmag1 = Reciprocal(vec_madd(s2,s2,vec_madd(c2,c2,vzero)));
c1 = vec_nmsub(s2, s2, vec_madd(c2, c2, vzero));
s1 = vec_madd(vtwo, vec_madd(s2, c2, vzero),
            vzero);
return vec_madd(s1, fixmag1, vzero);
}

```

When compiling on a PowerPC machine, the `-faltivec` option to `gcc` allows the construct

```
vector float vzero = (vector float)(0.);
```

This shorthand is not supported by Intel/SSE version of `gcc` so it may be necessary to manually edit the code to use the more general form

```
vector float vzero = {0.,0.,0.,0.};
```

The general form should work on both AltiVec and SSE versions of `gcc`, but in the above code a macro has been used to switch notations:

```

#ifdef ALTIVEC
#define VEC_CONST(x) (vector float)(x)

```

```
#else
#define VEC_CONST(x) {(x),(x),(x),(x)}
#endif
```

The benchmark results for this code, in machine cycles per array element are

Vector length	Linux/Intel	Linux/PPC
000256	29.7	27.1
001024	30.2	27.0
004096	30.3	27.1
016384	30.3	28.5
065536	31.9	28.5
131072	30.6	31.8

The timings in μ seconds:

Vector length	Linux/Intel	Linux/PPC
000256	3.9	4.6
001024	17.5	18.5
004096	61.7	74.1
016384	246.5	311.7
065536	983.7	1247.0
131072	1961.3	2779.5

9.3 Fourier Transform

To illustrate the conversion process on a typical FFT routine we use a 1024 point FFT module (a Stockham algorithm) which forms part of the N.A. Software multi-algorithm FFT suite. Results for this routine are:

	Linux/Intel	Linux/PPC
Cycles/elt	30.0	29.5
μ seconds	30.7	20.1

9.4 Rounding

The original `altivec2avx.h` only used instructions in SSE and SSE2 for greatest compatibility. However, SSE4.1 added `roundps`, `roundss`, *etc.* which can be used for `vec_round`, `vec_ceil`, `vec_floor` and `vec_trunc`. For instance the code

```
#include "altivec2avx.h"

vector float Avf32, Rvf32;
Avf32 = vec_ld(0, block_in1);
Rvf32 = vec_round(Avf32);
```

when analysed with `iaca.sh` (for just `vec_round`) gives

```
Total Throughput: 8 Cycles;          Throughput Bottleneck: Port1
Total number of Uops bound to ports: 31
Data Dependency Latency: 30 Cycles; Performance Latency: 34 Cycles
```

but replacing `vec_round(a)` with

```
_mm_round_ps(a1, _MM_FROUND_TO_NEG_INF)
```

gives

```
Total Throughput: 1 Cycles;          Throughput Bottleneck: Port1, Port2_ALU,
                                         Port2_DATA, Port3_ALU, Port4
Total number of Uops bound to ports: 4
Data Dependency Latency: 10 Cycles; Performance Latency: 11 Cycles
```

A significant improvement which is repeatable for `vec_ceil`, `vec_floor` and `vec_trunc`.

When available, the integer rounding functions of `altivec2avx.h` have been replaced with the SSE4.1 instructions:

```
vec_round  _mm_round_ps(x, _MM_FROUND_TO_NEAREST_INT)
vec_ceil   _mm_round_ps(x, _MM_FROUND_TO_POS_INF)
vec_floor  _mm_round_ps(x, _MM_FROUND_TO_NEG_INF)
vec_trunc  _mm_round_ps(x, _MM_FROUND_TO_ZERO)
```

10 AVX Considerations

This section discusses the extent to which the `altivec2avx.h` can be used on systems with the Advanced Vector Extension (AVX) instructions, and any performance benefits which may derive from AVX. We find that:

- The limitation of the AltiVec registers to 128 bits makes it very difficult within an include file such as `altivec2avx.h` to make use of the wider AVX registers.
- However, the `gcc` 4.4 compiler already provides substantial support for AVX. In particular, nearly all mappings provided in this version of `altivec2avx.h` are efficiently mapped by `gcc` onto the appropriate AVX instructions.
- The fused multiply-add AVX instruction is available in AltiVec, and when supported by AVX hardware, should be added.

10.1 AVX Features

Summarising the features of AVX is best done by quoting from http://en.wikipedia.org/wiki/Advanced_Vector_Extensions

1. *The size of the SIMD vector registers is increased from 128-bits XMM registers to 256-bits registers called YMM0 - YMM15.*

This option offers little advantage emulating AltiVec instructions as these are defined to be 128 bits by the API.

2. *Non-destructive instructions. The AVX instruction set allows all two-operand XMM instructions to be modified into non-destructive three-operand forms where the destination register is different from both source registers. For example `a:=a+b` is replaced by `c:=a+b` so that register `a` is unchanged after the instruction.*

This feature is examined in more detail in §10.3.

3. *The alignment requirement of SIMD memory operands is relaxed.*

This option may help, but for misaligned data AltiVec requires you to load two adjacent aligned vectors and extract the required elements. Emulating this misaligned access may be possible, but it will always be slow and, in general, it is better for the programmer to use the appropriate AVX load/store instructions.

10.2 AVX support in GCC

`gcc` 4.4 supports Intel AVX with the `-mavx` flag. This version of `gcc` requires `binutils` 2.19.51.0.1 or newer.

The header file `altivec2avx.h` does not directly include any SSE or AVX assembler instructions; instead we rely upon `gcc` intrinsics to generate the appropriate assembler. For

instance, the AltiVec instruction `vec_add` calls the function `_mm_add_ps` (for float arguments), which in turn, calls `__builtin_ia32_addps`. This then generates the appropriate SSE (`addps`) or AVX (`vaddps`) instructions depending upon whether `gcc` is passed the `-msse4` or `-mavx` flags.

Take a simple example:

```
#include "altivec2avx.h"

vector float Avf32, Bvf32, Rvf32;
Avf32 = vec_ld(0, block_in1);
Bvf32 = vec_ld(0, block_in2);
Rvf32 = vec_add(Avf32, Bvf32);
```

when compiled with a version of `gcc` that supports both SSE and AVX instructions sets

```
gcc-4.4 -O3 -S -flax-vector-conversions vadd.c -msse4
```

produces this assembler output:

```
movaps    (%edx), %xmm0
addps     (%eax), %xmm0
movl      %esp, %eax
andl      $-16, %eax
movaps    %xmm0, (%eax)
```

Using the same compiler and options, but targeting the AVX extensions:

```
gcc-4.4 -O3 -S -flax-vector-conversions vadd.c -mavx
```

produces

```
vmovaps   (%edx), %xmm0
vaddps    (%eax), %xmm0, %xmm0
movl      %esp, %eax
andl      $-16, %eax
vmovaps   %xmm0, (%eax)
```

`gcc` is generating the correct AVX instructions, though it is only using the 128 bit XMM registers. As a result of this feature of `gcc`, it is not necessary to insert explicit AVX instructions into the IA version of `altivec2avx.h`. However, we note again that given the 128 bit AltiVec API, the 256 bit YMM registers are of little direct use without manual recoding.

10.3 Register Pressure: Help from GCC

In `altivec2avx.h` we use the `gcc` intrinsics `vec_` which gives us some type protection, but also leaves the assignment of data to XMM registers up to the compiler. Obviously, the details of this will depend upon the internals of the compiler, but it is useful to ask the question. “Do the non-destructive instructions of AVX give the compiler more freedom to

assign results?”. We test this by considering some example codes (the codes themselves are listed in the Appendix).

10.3.1 4×4 Matrix Multiply

Appendix A gives AltiVec code for a 4×4 float matrix multiply. This was compiled with `altivec2avx.h` and the results analysed with the Intel Architecture Code Analyzer (`iaca.sh`) available from

<http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/>.

First, we tell `gcc` to just use SSE instructions:

```
gcc-4.4 -O3 -mtune=core2 -flax-vector-conversions vmatmul.c -msse4 && ./iaca.sh
-32 a.out
```

produces

Total Throughput: 43 Cycles;	Throughput Bottleneck: Port2_ALU, Port3_ALU, Port4
Total number of Uops bound to ports: 201	
Data Dependency Latency: 32 Cycles;	Performance Latency: 63 Cycles

Now if we tell the compiler to use the AVX instructions

```
gcc-4.4 -O3 -mtune=core2 -flax-vector-conversions vmatmul.c -mavx && ./iaca.sh
-32 a.out
```

produces

Total Throughput: 34 Cycles;	Throughput Bottleneck: Port2_ALU, Port3_ALU
Total number of Uops bound to ports: 169	
Data Dependency Latency: 32 Cycles;	Performance Latency: 53 Cycles

`gcc` is still using the 128 bit XMM registers, but with the AVX instruction set is able to use the non-destructive three-operand forms to gain performance.

Note that we have not modified `altivec2avx.h` nor the original source as we have only changed the compiler flags to `gcc`.

Performance could be further improved by use of a fused multiply-add operation which is used a lot in the matrix multiple (`vec_madd`). Currently `gcc` does not support this through its intrinsic functions (and nor does current AVX hardware) but in due course AVX assembler could be added to `altivec2avx.h` to support it.

10.3.2 Vector Sine

Taking the AltiVec algorithm to calculate the vector sine function given in Section 9.2 and compiling with

```
gcc-4.4 -O3 -mtune=core2 -flax-vector-conversions vsin.c -msse4 && ./iaca.sh
-32 a.out
```

produces

```

Total Throughput: 31 Cycles;          Throughput Bottleneck: Port1
Total number of Uops bound to ports: 102
Data Dependency Latency: 132 Cycles; Performance Latency: 136 Cycles

```

Now if we tell the compiler to use the AVX instructions

```
gcc-4.4 -O3 -mtune=core2 -flax-vector-conversions vsin.c -mavx && ./iaca.sh -32
a.out
```

produces

```

Total Throughput: 31 Cycles;          Throughput Bottleneck: Port1
Total number of Uops bound to ports: 91
Data Dependency Latency: 129 Cycles; Performance Latency: 135 Cycles

```

In this case, the improvement is slight. Many of the instructions require loading constants, limiting the compiler's options.

10.4 Benchmarks on AVX machine

All the previous results have been run on the Intel Architecture Code Analyzer (`iaca.sh`). We give here results run on a 1.6GHz Sandy Bridge 64-bit processor with 4 cores and 4096KB cache.

The test code was compiled with the command line:

```
gcc-4.4 -flax-vector-conversions -mavx -march=native -mfpmath=sse -O3
```

Timings were made by using the Time Stamp Counter `RDTSC` and by repeating each function 1000 times. The fastest number of clock cycles was then reported.

The results below were also repeated with `-msse4` instead of `-mavx`, but the results were found to be identical.

10.4.1 Inner product

An inner product function was implemented in C as

```
for (i = 0; i < 32; i++)
    res += a[i] * b[i];
```

Here `a` and `b` are arrays of 32 floats (32 bits). This same function was implemented in both AltiVec code, as given in Appendix B, and SSE code using

```
acc = _mm_add_ps(acc, _mm_mul_ps(A, B));
```

The AltiVec code also tested the new implementations of `vec_perm` and `vec_lvs1` which is commonly used for unaligned loads.

The results were:

	Cycles
Altivec	53
C	160
SSE/AVX	37

The SSE timings may be improved by using 256-bit AVX operations, but we needed to stay with the 128-bit API of Altivec.

10.4.2 4×4 matrix transpose

SSE provides a convenient macro for transposing a 4×4 matrix:

```
_MM_TRANSPOSE4_PS(vaT[0], vaT[1], vaT[2], vaT[3]);
```

Altivec, on the other hand, relies heavily on `vec_perm` to do the same operation

```
/* vec_perm() part 1 */
vtmp[0] = vec_perm(va[0], va[2], vpat1);
vtmp[1] = vec_perm(va[1], va[3], vpat1);
vtmp[2] = vec_perm(va[0], va[2], vpat2);
vtmp[3] = vec_perm(va[1], va[3], vpat2);

/* vec_perm() part 2 */
vaT[0] = vec_perm(vtmp[0], vtmp[1], vpat3);
vaT[1] = vec_perm(vtmp[0], vtmp[1], vpat4);
vaT[2] = vec_perm(vtmp[2], vtmp[3], vpat3);
vaT[3] = vec_perm(vtmp[2], vtmp[3], vpat4);
```

The timings are

	Cycles
Altivec	32
SSE/AVX	32

and indicate that we have an efficient implementation of `vec_perm`.

10.4.3 Multiply two 4×4 matrices

Appendix A gives an example of a 4×4 matrix multiply using Altivec instruction. As a SSE implementation was not readily available, the function was written in C in a way that would allow gcc's automatic vectorization to optimize the code:

```
for (i = 0; i < 4; i++) {
    float ai0=A(i,0), ai1=A(i,1), ai2=A(i,2), ai3=A(i,3);
    P(i,0) = ai0 * B(0,0) + ai1 * B(1,0) + ai2 * B(2,0) + ai3 * B(3,0);
    P(i,1) = ai0 * B(0,1) + ai1 * B(1,1) + ai2 * B(2,1) + ai3 * B(3,1);
    P(i,2) = ai0 * B(0,2) + ai1 * B(1,2) + ai2 * B(2,2) + ai3 * B(3,2);
    P(i,3) = ai0 * B(0,3) + ai1 * B(1,3) + ai2 * B(2,3) + ai3 * B(3,3);
}
```

The results are

	Cycles
Altivec	58
C	32

A 4x4 matrix multiply

Taken from http://developer.apple.com/hardwaredrivers/ve/algorithms.html#Matrix_Multiplication.

```
typedef vector float vFloat;

void MultiplyMatrix4x4(const vFloat *A, const vFloat *B, vFloat *C)
{
    //Load the matrix rows
    vector float A1 = vec_ld( 0, A );
    vector float A2 = vec_ld( 1 * sizeof( vector float), A );
    vector float A3 = vec_ld( 2 * sizeof( vector float), A );
    vector float A4 = vec_ld( 3 * sizeof( vector float), A );

    vector float B1 = vec_ld( 0, B );
    vector float B2 = vec_ld( 1 * sizeof( vector float), B );
    vector float B3 = vec_ld( 2 * sizeof( vector float), B );
    vector float B4 = vec_ld( 3 * sizeof( vector float), B );

    vector float zero = (vector float) vec_splat_u32(0);
    vector float C1, C2, C3, C4;

    //Do the first scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 0 ), B1, zero );
    C2 = vec_madd( vec_splat( A2, 0 ), B1, zero );
    C3 = vec_madd( vec_splat( A3, 0 ), B1, zero );
    C4 = vec_madd( vec_splat( A4, 0 ), B1, zero );

    //Accumulate in the second scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 1 ), B2, C1 );
    C2 = vec_madd( vec_splat( A2, 1 ), B2, C2 );
    C3 = vec_madd( vec_splat( A3, 1 ), B2, C3 );
    C4 = vec_madd( vec_splat( A4, 1 ), B2, C4 );

    //Accumulate in the third scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 2 ), B3, C1 );
    C2 = vec_madd( vec_splat( A2, 2 ), B3, C2 );
    C3 = vec_madd( vec_splat( A3, 2 ), B3, C3 );
    C4 = vec_madd( vec_splat( A4, 2 ), B3, C4 );

    //Accumulate in the fourth scalar x vector multiply for each row
    C1 = vec_madd( vec_splat( A1, 3 ), B4, C1 );
    C2 = vec_madd( vec_splat( A2, 3 ), B4, C2 );
    C3 = vec_madd( vec_splat( A3, 3 ), B4, C3 );
    C4 = vec_madd( vec_splat( A4, 3 ), B4, C4 );
}
```

```
//Store out the result
vec_st( C1, 0 * sizeof( vector float ), C );
vec_st( C2, 1 * sizeof( vector float ), C );
vec_st( C3, 2 * sizeof( vector float ), C );
vec_st( C4, 3 * sizeof( vector float ), C );
}
```

B Inner product

This implementation taken from <http://lists.xiph.org/pipermail/speex-dev/2004-January/000865.html>

```
static float inner_prod(float *a, float *b, int len)
{
    int i;
    float sum;

    int a_aligned = (((unsigned long)a) & 15) ? 0 : 1;
    int b_aligned = (((unsigned long)b) & 15) ? 0 : 1;

    __vector float MSQa, LSQa, MSQb, LSQb;
    __vector unsigned char maska, maskb;
    __vector float vec_a, vec_b;
    __vector float vec_result;

    vec_result = (__vector float)vec_splat_u8(0);

    if ((!a_aligned) && (!b_aligned)) {
        // This (unfortunately) is the common case.
        maska = vec_lvsl(0, a);
        maskb = vec_lvsl(0, b);

        MSQa = vec_ld(0, a);
        MSQb = vec_ld(0, b);

        for (i = 0; i < len; i+=8) {

            a += 4;
            LSQa = vec_ld(0, a);
            vec_a = vec_perm(MSQa, LSQa, maska);

            b += 4;
            LSQb = vec_ld(0, b);
            vec_b = vec_perm(MSQb, LSQb, maskb);
```

```
    vec_result = vec_madd(vec_a, vec_b, vec_result);

    a += 4;
    MSQa = vec_ld(0, a);
    vec_a = vec_perm(LSQa, MSQa, maska);

    b += 4;
    MSQb = vec_ld(0, b);
    vec_b = vec_perm(LSQb, MSQb, maskb);

    vec_result = vec_madd(vec_a, vec_b, vec_result);
}
} else if (a_aligned && b_aligned) {

    for (i = 0; i < len; i+=8) {
        vec_a = vec_ld(0, a);
        vec_b = vec_ld(0, b);
        vec_result = vec_madd(vec_a, vec_b, vec_result);
        a += 4; b += 4;
        vec_a = vec_ld(0, a);
        vec_b = vec_ld(0, b);
        vec_result = vec_madd(vec_a, vec_b,
                               vec_result);
        a += 4; b += 4;
    }

} else if (a_aligned) {
    maskb = vec_lvsl(0, b);
    MSQb = vec_ld(0, b);

    for (i = 0; i < len; i+=8) {

        vec_a = vec_ld(0, a);
        a += 4;

        b += 4;
        LSQb = vec_ld(0, b);
        vec_b = vec_perm(MSQb, LSQb, maskb);

        vec_result = vec_madd(vec_a, vec_b, vec_result);

        vec_a = vec_ld(0, a);
        a += 4;
```

```
    b += 4;
    MSQb = vec_ld(0, b);
    vec_b = vec_perm(LSQb, MSQb, maskb);

    vec_result = vec_madd(vec_a, vec_b, vec_result);
}
} else if (b_aligned) {
    maska = vec_lvsl(0, a);
    MSQa = vec_ld(0, a);

    for (i = 0; i < len; i+=8) {

        a += 4;
        LSQa = vec_ld(0, a);
        vec_a = vec_perm(MSQa, LSQa, maska);

        vec_b = vec_ld(0, b);
        b += 4;

        vec_result = vec_madd(vec_a, vec_b, vec_result);

        a += 4;
        MSQa = vec_ld(0, a);
        vec_a = vec_perm(LSQa, MSQa, maska);

        vec_b = vec_ld(0, b);
        b += 4;

        vec_result = vec_madd(vec_a, vec_b, vec_result);
    }
}

vec_result = vec_add(vec_result, vec_sld(vec_result, vec_result, 8));
vec_result = vec_add(vec_result, vec_sld(vec_result, vec_result, 4));
vec_ste(vec_result, 0, &sum);

return sum;
}
```