ARM NEON for C++ Developers

Contents

Preface

Some time ago, I've wrote an introduction to SIMD. It was an overview of SIMD instructions available on modern AMD64 processors. Since I wrote that article, I happened to spend some time writing vectorized ARM NEON code. I thought it's a good idea to write comparable article for NEON.

All the code I have written was for embedded Linux platforms, specifically for customized Debian, and built with GCC. The stuff is not specific to Linux at all, and most parts of it is not specific to GCC either. Should be useful for people programming for Android, iOS, ARM Windows, or even bare metal ARM.

I'll try to write the article in a way so the prior experience with AMD64 SIMD is not required. For this reason, I gonna copy-paste some parts of that prior article, and reuse the overall structure.

The scope is more or less the same, a starting point with an overview of what's available.

Hardware-wise, the focus is on ARM v7 instruction set. That instruction set is supported by 32-bit Debian running on Rockchip and Broadcom SoCs I've developed for. It's very compatible otherwise, supported by Raspberry Pi starting from Pi2, iPhone starting from 3GS, all versions of iPad, vast majority of Android devices made after 2011, and majority of modern ARM SoCs, even very modestly priced $3-4 Allwinner chips.

Introduction to SIMD

The acronym stands for "single instruction, multiple data". In short, it's an extension to the instruction set which can apply same operation onto multiple values. These extensions also define extra set of registers, wide ones, able to hold these multiple values in a single register.

For example, imagine you want to compute squares of 4 floating point numbers. A straightforward implementation looks like this:

```
void mul4_scalar( float* ptr )
{
	for( int i = 0; i < 4; i++ )
	{
		const float f = ptr[ i ];
		ptr[ i ] = f * f;
	}
}
```

Here's a NEON version which does exactly same thing:

```
void mul4_vectorized( float* ptr )
{
	float32x4_t f = vld1q_f32( ptr );
	f = vmulq_f32( f, f );
	vst1q_f32( ptr, f );
}
```

Unless C++ optimizer does a good job at automatic vectorization[1], the scalar version compiles into a loop. That loop is short, so branch prediction fails 25% of iterations. The scalar code will likely contain some loop boilerplate, and will execute the body of the loop 4 times.

The vectorized version contains 3 instructions and no loop. No branches to predict, very straightforward machine code that's very fast to execute.

ARM is used a lot in battery-powered devices like phones and tablets. I haven't measured current, but I would expect SIMD instructions to be more power-efficient than equivalent scalar code, because less instructions to execute, less RAM requests to fulfill, and most importantly because less wall clock time to run and the CPU will go back to sleep sooner: mobile CPUs do that kind of power scaling really fast.

Documentation

Here's 2 main links I have in my bookmarks.

Neon Intrinsics page on arm.com is useful when you know the exact intrinsic you want, or can guess the beginning of name, and want to know what it does. When you use that, don't forget to check the instruction set field, some intrinsics are only available for A32/A64 but not for ARM v7.

Compiler Reference is useful to find what's available.

Unfortunately, I don't know where to get instruction timings, latency and throughput. Apparently, the information is unavailable whatsoever.

Conventions

In many places of this article, you'll see things like this: `vmax[q]_<type>`. It's a shortcut for a group of intrinsics who compile into similar instructions.

The q is optional, may or may not be present. When present, the instruction operates on 16-byte vectors; when missing, it handles 8-byte vectors.

The <type> can be any of this: s8, u8, s16, u16, s32, u32, s64, u64, f32; `s` prefix is for signed integer lanes, `u` prefix for unsigned integer lanes, `f` for floats, and the number after the prefix is count of bits per lane.

See the list on that page for these particular vmax[q]_<type> intrinsics. For example, vmaxq_u16 computes maximum of uint16_t values, operates on 16-byte registers so both inputs and output contain 8 uint16_t lanes.

Data Types

Most of the time you should be processing data in registers. The ideal pattern for SIMD code, load source data to registers, do as much as you can while it's in registers, then store the results into memory. The best case is when you don't have a result, or it's very small value like bool or a single scalar, there're instructions to copy values from SIMD registers to general purpose ones.

[1] In practice, they usually do a decent job for trivially simple examples like the above one. They usually fail to auto-vectorize anything more complex. They rarely do anything at all for code which operates on integers, as opposed to floats or doubles.

NEON has many built-in data types for SIMD vectors. This is unlike AMD64, where SIMD vectors only have 6 fundamental types, 3 of which are 32-bytes long only used with AVX instructions.

16-byte vectors

The code snippet in the intro section uses *float32x4_t* data type. The type representing 16-bytes vector containing 4 single-precision floating point values. NEON has many more of them with integers of various size like int32x4_t, int16x8_t or *uint8x16_t* (all integer types are available, 8, 16, 32 or 64 bits, signed or unsigned) and even half-precision floats, *float16x8_t*.

8-byte vectors

Unlike AMD64, NEON instructions have versions processing 8-byte SIMD vectors. They handle different data types, e.g. *float32x2_t* for 8-byte vector containing 2 single-precision floating point values. That particular *float32x2_t* data type is extremely useful to implement algorithms in the domain of 2D graphics, as you can interpret the vectors as a 2D position or offset.

Integer and half-precision float vectors also have 8-byte versions.

Multiple vectors

Some intrinsics use larger SIMD types composed of multiple vectors in a single variable. An example of such type is int16x8x4_t: it holds 64 bytes of data in total, that data is interpreted as 4 SIMD vectors, each vector contains 8 scalars, int16_t each i.e. they are signed 16-bit integers. In C, that thing is a structure containing an array of int16x8_t types but not in assembly, in assembly the whole thing is in registers. Most notable use of these data types is interleaved load/store, but some other instructions do that as well, when they return two vectors instead of a single one.

Moving and Converting SIMD Vectors

Reinterpret Cast

Sometimes you want to convert between the types without changing bits in the registers. In C there're many intrinsics to accomplish that, the exact names depend on the data type. I won't repeat what's written on arm.com, here's a link instead.

You can use them to cast stuff as long as source and destination vectors have the same size in bytes, including integers to floats or vice versa. They don't change bits in the registers, e.g. vreinterpretq_u32_f32 will convert 1.0f float into 0x3f800000 value.

Converting Types

Again, I won't repeat the documentation, here's a link.

These instructions don't change count of lanes, they transform each lane of the source into a corresponding lane of the result.

Converting to/from floats

Float-to-integer conversions use rounding toward zero.[2]

Integer-to-float conversions use rounding mode from FPCR register, the default appears to be "to nearest". If you really want to change that use fesetround from C runtime library, but beware of

[2] At least according to the documentation, I have not tested the actual behavior.

potential issues. Compiler may reorder stuff. Also changing that register may break totally unrelated code running on the same native thread.

The float conversation instructions which have _n_ in the name convert to/from fixed point integers, e.g. vcvtq_n_s32_f32 with the last argument 8 will convert 1.0 float into 0x100 integer, i.e. it will use 24.8 fixed point format.

Converting between integers

The "narrow integer" means convert each lane into half the width, discarding the most significant byte[s] of each lane.

The "saturating narrow integer" means convert each lane into half the width, clipping out of range values to min/max of the destination lane data type.

Finally, "long move" means expand each lane to twice the source size, filling most significant byte[s] of the results with zeros (unsigned version) or with the sign bit of the source value (signed variants).

Split and Combine

Use vcombine_<type> to make one 16-byte vector out of two 8-byte ones.

Use vget_low_<type> and vget_high_<type> for the opposite.

In assembly, both 16 and 8-byte vectors are in the same registers. Moreover, both lower and higher halves of 16-byte vectors are directly addressable in NEON. This means in many cases split and combine intrinsics compile into no instructions, i.e. they are often free performance-wise.

Inserting and extracting scalars

There're many intrinsics which insert a scalar value into vector, or do the opposite. The extract ones are vget[q]_lane_<type>, the inserts are vset[q]_lane_<type> and they do what you'd expect, get/set a single lane of a vector.

Note the lane index is encoded into the instruction. This means a `for` loop won't compile, the index needs to be constexpr expression, like an integer template argument in C++.

Memory Access

The documentation for them is scattered around "Loads of a single vector or lane", "Store a single vector or lane" and "Loads of an N-element structure" section of the documentation.

To load/store a complete vector, use vld1[q]_<type>/ vst1[q]_<type> intrinsics. Using the variant with the `q` will cause 16-byte vectors to be loaded/stored, otherwise it will do 8-byte vectors.

NEON supports broadcast loads (just like AVX), where a single lane is loaded from memory and broadcasted into all lanes of the result. These are vld1[q]_dup_<type> intrinsics.

There're intrinsics to load/store a single lane from a vector. However, doing so in performance-critical paths can be slow. If you need to do that often, consider reworking RAM layout of your stuff. Accessing memory in larger blocks, ideally in complete vectors, is faster in general.

Interleaved RAM access

That's a unique and very useful feature of NEON. Let's say you have a pointer to densely packed 3D vectors in memory, each vector with 3 float fields for X, Y and Z coordinates. You can use vld3q_f32 to load 4 such structures into 3 SIMD registers, one with all 4 X values, another one with all 4 Y values, and the third one with Z values. Works for stores too, if you have 3 SIMD vectors and want to store the lanes interleaved, use vst3q_f32 intrinsic.

Interleaved load/store instructions support all scalar types even bytes, but only 2, 3 or 4 vectors being [de]interleaved. If you want more vectors, you'll have to do something else instead.

This works for both floats and integers. For example, vld3q_u8 is the fastest way to load and de-interleave 24-bit RGB pixels; a single instruction will load 16 pixels into 3 vector registers, one per channel. Similarly, vld2q_s16 can be used to load 8 samples of 16-bit PCM stereo, splitting into channels.

Initializing Vector Registers

All vector types have intrinsics to broadcast scalar variable into all lanes of the result, here's the documentation. Apparently, each instruction is exposed as 2 intrinsics, e.g. vdup_n_u8 and vmov_n_u8. I have no idea why, could be just legacy.

Initializing a SIMD vector with different values is rather tricky, here's the docs. Unfortunately, these intrinsics are not usable as they are. One way to implement an equivalent of `_mm_setr_ps` SSE intrinsics is below:

```
 // https://stackoverflow.com/a/49459143/126995
inline uint32_t float_bits( const float f )
{
	union
	{
		uint32_t u;
		float f;
	}
	temp;
	temp.f = f;
	return temp.u;
}
inline float32x2_t vector2set( float x, float y )
{
	const uint32_t low = float_bits( x );
	const uint32_t high = float_bits( y );
	uint64_t combined = high;
	combined = combined << 32;
	combined |= low;
	return vcreate_f32( combined );
}
inline float32x4_t XMVectorSet( float x, float y, float z, float w )
{
	return vcombine_f32( vector2set( x, y ), vector2set( z, w ) );
}
```

The above code bit-casts floats into integers, creates *`uint64_t`* variable with 2 of them, converts to SIMD vectors, then combines two 8-byte vectors into the result.

Bitwise Instructions

ARM documentation calls them "logical operations".

Unlike AMD64 there're dedicated bitwise "not" instruction, and bitwise "or not" too, called "Bitwise OR Complement" in the docs. An equivalent of SSE's bitwise `andnot` is also there, called "bit clear". Bitwise XOR is called "EOR", for Exclusive OR.

Bitwise Select

This instruction takes 3 arguments, and computes the following expression, for each bit in the vector independently: `( a ) ? b : c`

Here's an example where it can be useful. It assumes the input vector was made by a comparison intrinsic like vcltq_f32 which sets each lane to either zero or UINT_MAX (all ones).

```
extern const __attribute__((weak)) uint32x4_t g_positionValues { 0, 1, 2, 3 };
// Index of the first non-zero lane in the input vector
// Returns 4 if the vector doesn't have any UINT_MAX lanes
inline uint32_t firstTrueIndex( uint32x4_t cmp )
{
    // Replace UINT_MAX with corresponding lanes from g_positionValues, and zeroes with 4
    const uint32x4_t falseValues = vdupq_n_u32( 4 );
    const uint32x4_t selected = vbslq_u32( cmp, g_positionValues, falseValues );
    // Compute minimum value of the halves
    const uint32x2_t selected2 = vmin_u32( vget_low_u32( selected ), vget_high_u32( selected ) );
    // Return minimum value of both lanes of the above
    return vget_lane_u32( vpmin_u32( selected2, selected2 ), 0 );
}
```

Floating Point Instructions

On ARM v7, NEON only supports single-precision 32-bit floats.

There's an awesome library written by Microsoft which supports NEON, DirectXMath. Whenever you need stuff like dot products, cross products, vector-matrix multiply, or similar primitives widely used in 3D or 2D graphics, no need to write and debug your own version, copy-paste from there. MIT license allows that.

Because MS code is portable across SSE and NEON, it doesn't use 8-byte long vectors, only 16-bytes long. For optimal performance on 2D vectors, you gonna need to switch data types to *float32x2_t* and remove `q` from intrinsics.

Arithmetic

Traditional

In ARM v7, only add, subtract and multiply are there. vdivq_f32 is only appeared in A64. If you need to divide floats on ARM v7, copy-paste a few lines from XMVectorDivide library function.

NEON has instructions to negate values, vneg[q]_f32, and compute absolute, vabs[q]_f32, no need to use bit tricks with -0.0f vector, like we have to do for SSE.

There's no square root instruction but there's an approximate version, vrsqrte[q]_f32. If you need more precision, copy-paste XMVectorSqrt from DirectXMath, just like for divisions, MS did Newton-Raphson refinement there.

Finally, there's an intrinsic to multiply SIMD vector by a scalar, vmul[q]_n_f32. The scalar is taken from a float variable. There's very similar one vmul[q]_lane_f32 where the scalar is from a lane of another vector.

Unorthodox

There're instructions for min/max just like in SSE, in NEON they are vmin[q]_f32 and vmax[q]_f32.

There's an instruction for absolute difference, vabd[q]_f32.

There're instructions to compute approximations of $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$, they are vrecpe[q]_f32 and vrsqrte[q]_f32 respectively.

Comparisons

There're all-lanes versions of equality comparison and the rest of them, <, >, ≤, ≥. The only missing one is inequality, ≠. Moreover, there're instruction to compare absolute values in two vectors. These intrinsics return `uint32x4_t` vector with lanes either 0 or all ones, which is UINT_MAX.

There's no equivalent of _mm_movemask_ps or _mm_testz_ps. If you need results in a scalar register to be able to write `if( something )` in C++, you'll have to implement manually with integer NEON intrinsics.

Horizontal Operations

There's vpadd_f32 intrinsic that takes two vectors [a, b] and [c, d], and returns [a+b, c+d]. No 16-byte version is available, only the 8-byte version. A useful application is final reduction step of dot product.

Same with horizontal pairwise minimum and maximum, the instructions are vpmin_f32 and vpmax_f32, the documentation calls them "folding minimum / maximum".

Shuffles

Unlike SSE, there's no generic shuffles. Whenever you need it, you'll have to do something else instead. Here's what NEON can do for you.

- Split then combine back. This way you can shuffle [a, b, c, d] into e.g. [c, d, c, d] or [c, d, a, b].
- Concatenate two vectors and cut a piece of the merged sequence, vextq_f32 does that. When the 2 input vectors are [a, b, c, d] and [e, f, g, h], it can return [b, c, d, e], [c, d, e, f] or [d, e, f, g].
- Broadcast a single lane into all lanes of the output with vdupq_lane_f32. But note some arithmetic intrinsics like vmul_laneq_f32 use scalar operand from arbitrary lane of a vector, you don't need broadcast for that.
- Interleave elements. The vzipq_f32 takes 2 vectors [a, b, c, d] and [e, f, g, h], returns other 2 vectors with [a, e, b, f] and [c, g, d, h].
- De-interleave elements. vuzpq_f32 takes 2 vectors [a, b, c, d] and [e, f, g, h], returns other 2 vectors with [a, c, e, g] and [b, d, f, h]
- Swap pairs of elements: vrev64q_f32 takes [a, b, c, d] and returns [b, a, d, c].
- Transpose elements.
  This instruction views first vector as the top rows of a sequence of 2x2 matrices, and second vector as bottom rows of corresponding matrices. It transposes all matrices stored this way, and returns 2 new vectors with transposed 2x2 matrices.
  Specifically for floats, vtrnq_f32 takes 2 vectors [a, b, c, d] and [e, f, g, h], returns other 2 vectors with [a, e, c, g] and [b, f, d, h]
- You can move individual lanes with vsetq_lane_f32 and similar.

- As a last resort, there's table lookup instructions. See this SO answer for an example, BTW Chuck Walbourn who wrote that answer is the main developer of DirectXMath library.

For shuffle intrinsics accepting 2 values, note that you can pass same vector to both arguments. This way vextq_f32 can shuffle [a, b, c, d] into [b, c, d, a], [c, d, a, b] or [d, a, b, c] vectors, vtrnq_f32 will shuffle into [a, a, c, c] and [b, b, d, d], vzipq_f32 into [a, a, b, b] and [c, c, d, d], and vuzpq_f32 into [a, c, a, c] and [b, d, b, d].

Fused Multiply-Add

FMA instructions take 3 arguments, each being a vector of 32-bit floating point numbers, a, b and c, and compute $(b \cdot c) + a$. NEON doesn't call them FMA though, they call it MLA for Multiply Accumulate but it's the same thing.

vmla[q]_f32 computes $(b \cdot c) + a$, and vmls[q]_f32 computes $a - (b \cdot c)$.

There're also versions of them which take c from scalar float variable, or from a specified lane of a vector register.

Integer Instructions

Arithmetic

All of them are supported for all lane sizes, except 64-bit lanes for most instructions.

Traditional

Additions and subtractions do what you would expect, e.g. `vadd[q]_s32` adds 32-bit lanes of two integer registers. Saturated versions are available as well, for int32_t lanes that's vqaddq_s32.

Unlike SSE, multiply support is implemented for all lane sizes.

Just like SSE, there's no integer divide instructions, but there're many bit shifting ones.

Unorthodox

For all lane sizes except 64-bit ones there're integer minimum and maximum instructions, e.g. vmin[q]_u8 for unsigned bytes.

Also for absolute difference of 2 vectors, vabd[q]_<type>. There're also accumulating versions of these, vaba[q]_<type>, and versions with accumulator twice as wide, vabal_<type>.

Unlike AMD64, NEON has many instructions for integer FMA, both truncating and accumulating versions are available. There're also instructions who use accumulator with wider lanes, here's an example:

```
int32x4_t vmlal_s16( int32x4_t a, int16x4_t b, int16x4_t c );
```

That instruction uses 32-bit accumulator to accumulate results of 16-bit multiplication.

There're some stranger ones, e.g. vqrdmulhq_s16 multiplies the values, doubles the results, and returns the most significant half of it.

Comparisons

Neon supports all lane types for them except 64-bit ones, both signed and unsigned versions.

Another thing, there's vtst[q]_<type> instruction. It computes this for each lane:

```
( ( a & b ) != 0 ) ? ALL_ONES: 0;
```

The ALL_ONES in the above expressions is the vector filled with all ones, -1 for signed lane types or maximum value for unsigned lane types. You can supply same value into both arguments, in this case it will just compare lanes for non-zero. You can do that with regular comparisons, but unlike vtst you'll need to supply a vector filled with zeros.

Bit Shifts

There're many instructions for that.

The normal shifts are equivalents of C++ operators, e.g. vshlq_n_s32 does the equivalent of a << b for uint32_t lanes where b is compile-time constant, vshlq_u32 does the same where b comes from corresponding lane of another vector.

Rounding right shifts increment (positive) or decrement (negative) result by 1 when the most significant of the discarded bits was 1.

There're many more shift intrinsics there, see Shifts by signed variable, Shifts by a constant, and Shifts with insert sections of the reference. The insert version is really strange, here's a scalar equivalent of vsriq_n_u32 instruction:

```
std::array<uint32_t, 4> vsriq_n_u32( std::array<uint32_t, 4> a, std::array<uint32_t, 4> b, int n )
{
    const uint32_t leftoverMask = UINT_MAX << ( 32 - n );
    std::array<uint32_t, 4> result;
    for( int i = 0; i < 4; i++ )
    {
        const uint32_t shifted = b[ i ] >> n;
        result[ i ] = ( a[ i ] & leftoverMask ) | shifted;
    }
    return result;
}
```

Horizontal Operations

Just like floats, integers have horizontal pairwise add, and horizontal pairwise min/max. These are vpadd_<type>, vpmin_<type> and vpmax_<type>. On ARM v7, these can only handle 8-byte long vectors, the 16-byte versions require A64 instruction set.

I often use them (followed by vgetlane_<type>) when I need to reduce vectors to scalars.

Shuffles

Again, no generic shuffles. However, everything I wrote above for floats also applies to integer vectors.

Just one change there, the "Reverse vector elements" family of intrinsics can reverse more than just pairs. vrev64q_u16 takes a vector of 16-bit values [a, b, c, d, e, f, g, h] and returns [d, c, b, a, h, g, f, e]

Other Integer Instructions

Absolute and Negate

Just like for floats, there're instructions to compute absolute and negate values. Unlike floats, the range of signed integers is asymmetric, e.g. signed bytes range is [-128 .. +127], both negate and absolute will overflow for some inputs. For this reason, they come in traditional and saturated versions. vqabsq_s8 is saturated, will transform -128 into +127. Non-saturated version, vabsq_s8, will not change -128.

Population Count

NEON has vector version of population count, vcnt[q]_u8 and vcnt[q]_s8. It computes population count of each byte of the input, and returns a vector with byte values in [0 .. 8] interval. If you need population count of 32-bit lanes you'll need to add groups of 4 bytes together. Here's one way to accomplish that.

```
inline uint32x4_t vcntq_u32( uint32x4_t v )
{
    // Compute popcnt for each byte of the input.
    uint8x16_t popcnt = vcntq_u8( vreinterpretq_u8_u32( v ) );
    // Add them, flipping sequences of 2 bytes.
    popcnt = vaddq_u8( popcnt, vrev16q_u8( popcnt ) );
    // Add them, flipping sequences of 4 bytes.
    popcnt = vaddq_u8( popcnt, vrev32q_u8( popcnt ) );
    // Now the vector has same byte in every 4-bytes long sequence.
    // The shift discards 3 out of the 4 bytes, and fills 3 higher ones with zeros.
    return vshrq_n_u32( vreinterpretq_u32_u8( popcnt ), 24 );
}
```

Bit Scan

NEON has vector versions of bit scan instructions. vclz[q]_<type> returns count of leading zeroes in lanes. vcls[q]_<type> returns count of leading sign bits in signed integer lanes.

Random Tips and Tricks

First and foremost, it doesn't matter how fast you crunch your numbers if the source data is scattered all over the address space. RAM access is very expensive these days, a cache miss can cost hundreds of CPU cycles. Even L1 cache is noticeably slower to access than registers. Keep your data structures SIMD-friendly. Prefer `std::vector` or equivalents like `eastl::vector` over the rest of the containers. When you read them sequentially, prefetcher part of the CPU normally hides RAM latency even for very large vectors which don't fit in caches. If your data is sparse, you can organize it as a sparse collection of small dense blocks, where each block is at least 1 SIMD register in size.

There're good vectorized libraries for C++ with NEON support: Eigen, DirectXMath mentioned earlier, couple others. Learn what they can do and don't reinvent wheels. They have lots of complicated stuff already implemented and debugged. If you enjoy looking at scary things, look at the source code of NEON version of XMVector3TransformCoordStream library routine.

Intrinsics are really fast, each one compiles into a single instruction. This means your lower-level SIMD routines are better to be inline functions in headers or *.inl files, otherwise you're at the mercy of the optimizer to inline them. I typically define and use this macro in GCC to improve my chances:
```
#define __forceinline inline __attribute__((always_inline))
```
Doesn't mean your whole code needs to be in headers, at some level of abstraction it's OK to place implementations in *.cpp files to improve modularity and compilation speed.

Don't write `static const uint32x4_t x = something();` inside functions or methods. In modern C++ that construction is guaranteed to be thread safe. To support the language standard, a compiler has to emit some boilerplate code, which gonna have locks and branches. Instead, you can place that value in a global variable so they're initialized before `main()` starts to run, or for DLLs before `dlopen` returns.

Sometimes it's a good idea to create C++ classes with SIMD vectors inside. If you have a class with a couple of vectors, create it on stack, and never create references nor pointers to it, compiler normally places these fields in SIMD registers. This way there's no class anywhere in machine code, no RAM loads or stores, and the performance is good. There's still a class in C++, when done right, it makes code easier to work with and reason about. Same applies to small `std::array`-s of SIMD vectors, or C arrays, unless you use non-constexpr variables to index these arrays.

It sometimes makes sense to reorder or even duplicate data in vectors, to optimize for runtime performance. I once needed to transform many 2D vectors with a 3x2 matrix, here's what I did.

```
// 3x2 matrix in SIMD registers. Optimized for batch processing use case, when you have a whole lot
of 2D vectors to transform.
struct SimdMatrix
{
    // [ m11, m22, m11, m22 ]
    float32x4_t matrixDiagonal;
    // [ m21, m12, m21, m12 ]
    float32x4_t matrixSides;
    // [ dx, dy, dx, dy ]
    float32x4_t add;
    // ..skipped some stuff, most notably constructors
    inline float32x4_t transformTwoPoints( float32x4_t points ) const
    {
        // [ dx + p.x * m11, dy + p.y * m22 ]
        const float32x4_t central = vmlaq_f32( add, points, matrixDiagonal );
        // Shuffle into [ p.y, p.x ]
        const float32x4_t flipped = vrev64q_f32( points );
        // [ central.x + p.y * m21, central.y + p.x * m12 ]
        return vmlaq_f32( central, flipped, matrixSides );
    }
};
```

The data layout looks weird and inefficient (3 vector registers to store just 6 scalars), but this way the multiplication only takes 3 instructions for 2 points, which was the only important use case for that code. This snippet also illustrates why FMA is awesome, even when called MLA[3].

Arithmetic instructions operating on half precision 16-bit floats only appeared in A32 and A64. If your target platform is ARM v7, the only thing you can do with them is convert to/from 32-bit float vectors, just like on AMD64 with FP16C. For this reason, I have only used these types to prepare data for GPU: modern mobile GPUs like Mali or Broadcom support GLES 3.0 and therefore half-floats.

Linux Specific

The Linux header with all these intrinsics and related shenanigans is `<arm_neon.h>` I think it's shipped with GCC. Here's cmake compiler options I use to build my code:

```
target_compile_options( ProjectName PUBLIC -Wall -Wno-psabi -Wno-comment -
march=native -mfpu=neon-fp16 -mfp16-format=ieee -ffast-math -O3 -fPIC )
```

I've learned everything I know about NEON assembly by looking at what GCC did to my C++ code. Here's a Linux shell command to disassemble a DLL:

```
objdump -d somelib.so > somelib.asm
```

Then search for a function name or other symbol.

[3] However, the implementation sacrifices a bit of accuracy for the performance. Due to the way floats are stored, it's slightly more accurate to compute flip/rotate/scale first, and apply translation as a last step.