SIMD for C++ Developers

Contents

Introduction

Motivation

I've noticed many programmers I'm working with aren't familiar with SIMD. I don't want to stop writing vectorized code, the performance is just too good. Instead I'm writing this article hoping to educate people.

Scope

Reading this article, or any article at all, won't make you an expert in the technology. I believe programming is not science but an applied skill. One can't become good in it unless practicing a lot. This article is an overview and a starting point.

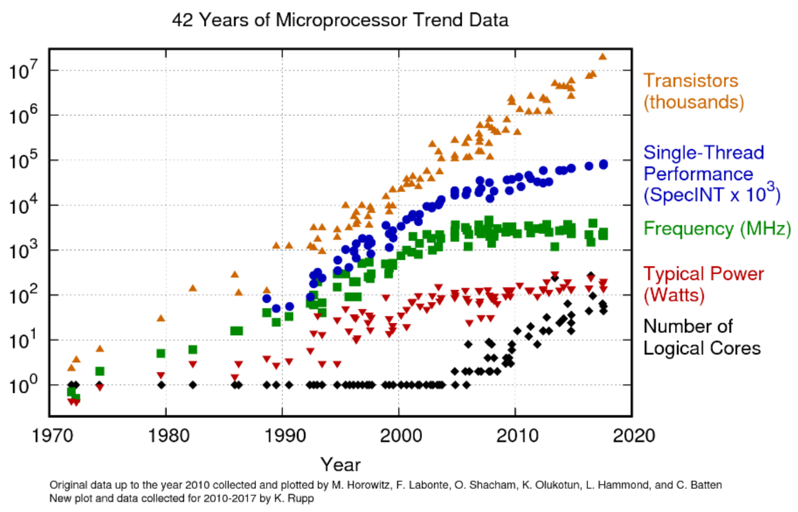
Most of this article is focused on PC target platform. Some assembly knowledge is recommended, but not required, as the main focus of the article is SIMD intrinsics, supported by all modern C and C++ compilers. The support for them is cross-platform, same code will compile for Windows, Linux, legacy OSX (before ARM64 M1 switch), and couple recent generations of game consoles (except Nintendo which uses ARM processors).

History Tour

At some high level, a processor is a mechanism which executes a sequence of instructions called "program". Specifically, it runs an endless loop with the following steps.

1. Fetch an instruction at the current instruction pointer.
2. Advance the current instruction pointer so it points to the next instruction in the program.
3. Execute the instruction. There're different possibilities here, an instruction can do math on values in registers like `inc rax`, access memory like `mov rax, qword ptr [rcx]`, do input or output[1], or mess with the current instruction pointer, implementing jumps, loops or procedure calls. PC CPUs can do many of these with a single instruction, `and dword ptr [r8], 11` does both load, bitwise and, and store.
4. Go to step 1.

Before 2000, the best thing to do when designing new processors was increasing clock frequencies. Then CPU designers hit multiple major obstacles, and were no longer able to do so. On the right, there's a nice picture I've found on the Internets. CPU manufacturing technology was and still is advancing fast, the CPU designers needed to come up with some new ideas how to use all these extra transistors to make faster processors, without being able to increase clock frequency.



[1] Modern PC programs almost never do I/O, because nowadays only OS kernel and drivers have direct access to hardware. Instead, they call some OS kernel API, which calls a device driver, which finally does the I/O.

One obvious approach is making more processors, i.e. increasing cores count. But many programs are not that parallel. And in modern world, these who are *that* parallel don't run on CPUs, GPGPUs are more suitable for them and have way more compute power.

Another approach is increasing size of CPU caches, but that improvement has diminishing returns, at some point program speed no longer limited with memory, you actually need to compute more stuff.

Another approach (very complicated, nevertheless widely used in modern processors), running multiple instructions per clock with instruction-level parallelism, deep pipelining, speculative execution. Newer CPUs do it progressively better, but just like cache sizes this approach has diminishing returns. With deep pipelines, when processor mispredicts a branch, too much time is wasted, also too much stuff is required to rollback results of the false-started instructions.

In addition to the above, CPU designers wanted to compute more stuff with same count of instructions. That's why they have invented SIMD.

Modern SIMD processors entered mainstream market with the release of Pentium III in 1999, and they never left. Technically MMX and 3DNow! were before that and they are SIMD, too, but they are too old, no longer relevant for developers.

Even cell phones support SIMD now, the instruction set is called ARM NEON. But this article is focused on PC platform, so NEON is out of scope.

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 vectorized SIMD version which does the same thing:

```
void mul4_vectorized( float* ptr )
{
	__m128 f = _mm_loadu_ps( ptr );
	f = _mm_mul_ps( f, f );
	_mm_storeu_ps( ptr, f );
}
```

Unless C++ optimizer does a good job at automatic vectorization[2], 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.

[2] In practice, they usually do a decent job for trivially simple examples like the above one. They often 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.

The vectorized version contains 3 instructions and no loop. No branches to predict, very straightforward machine code that's very fast to execute. By the way, when CPU vendors brag about stuff like "32 single-precision FLOPs/cycle", what they actually mean, you'll get that performance only if you'll vectorize the hell out of your code, using maximum SIMD width available on the hardware. More information is available in FMA section of this document.

The above example only requires SSE 1, will run on your grandmother's Pentium III. SSE 1 operates on 16 bytes registers. In C and C++, these registers are exposed as `__m128` data type: 128 bits = 16 bytes. Each register contains 4 single-precision floating point numbers, and the instruction set has 8 of these registers, they are named `xmm0` to `xmm7`.

SSE 2, introduced with Pentium 4 in 2000, added support for double-precision and integer SIMD instructions. They process same 8 registers, but for type safely these are exposed in C as `__m128d` and `__m128i` data types, respectively. Their length stayed the same 16 bytes, so there're just 2 double lanes there, compared to 4 single-precision float lanes in SSE 1.

When AMD introduced their AMD64 architecture in 2003, they have incorporated SSE 2 as a part of their then-new instruction set. I'll repeat it bold: **every 64-bit PC processor in the world is required to support at least SSE 1 and SSE 2**. At the same time, AMD added 8 more of these vector registers, the new ones are named `xmm8` to `xmm15`.

On modern PCs, the single-lane version takes exactly same time to execute as the wide version. The reason single-lane arithmetic instructions exist, it wasn't always the case. Pentium III didn't have 128-bit wide arithmetic units, it did 128-bit math by splitting lanes into 64-bit pieces, and computing them independently. The same way previous generation AMD Ryzen 1000- and 2000-series didn't have 256-bit wide arithmetic units, to execute AVX and AVX2 instructions they split the operands into 128-bit pieces.

The mandatory support of SIMD opened a few possibilities. Modern compilers don't usually emit old-school x87 instructions when compiling 64-bit code which does math on float or double numbers. Instead, the compilers emit SSE 1 and SSE 2 instructions. The floating-point math instructions have single-lane versions. The above snippet uses *_mm_mul_ps* intrinsic (compiles into `mulps` instruction) to compute squares of 4 numbers. SSE 1 has matching instruction that only multiplies one lowest lane, *_mm_mul_ss*, it compiles into `mulss` instruction. The upper 3 lanes are copied from the first argument without multiplying them.

Apparently, CPU vendors viewed SIMD as a success, because they continued adding new vector instructions. SSE 3, SSSE 3 and SSE 4.1 all added new instructions operating on these SIMD registers.

By now, these instruction sets are almost universally supported by mainstream CPUs. At the time of writing, steam hardware survey says 100.00% of users have SSE3, and 98.76% of them have SSE 4.1; expand "other settings" panel to see these numbers on that web page.

Then in 2011 Intel introduced AVX in their Sandy Bridge processors. That version extended 16-byte registers into 32 bytes, so they now can hold 8 single-precision floats, or 4 doubles. Their count stayed the same, 16 registers. In assembly, new registers are named `ymm0` to `ymm15`, and their lower 128-bit halves are still accessible as `xmm0` to `xmm15`. The same way the lowest byte of `rax` register is accessible as `al`. AVX supports operations on 32- and 64-bit floats, but not much for integers.

Then in 2013 Intel introduced AVX 2. No new registers, but they have added integer math support there. Both AVX and AVX 2 are widely supported by now, but not yet universally so on desktops. See hardware support section for details.

Intel tried to proceed with AVX 512 but this time it didn't work out. Maybe 32 bytes vector width is enough already. Maybe it takes too many transistors to implement or too much electricity to run. Maybe Intel was too greedy and asked too much money for IP license.

Regardless on the reason, very few modern processors support AVX 512, all of them made by Intel. At the time of writing, Q1 2021, I think AVX 512 is safe to ignore. All consoles and 99% of PCs don't support AVX 512. Current AMD chips are much faster than Intel, AVX 2 code running on AMD CPU will likely be faster than AVX 512 code running on a comparable Intel CPU. For this reason, this article covers SIMD up to AVX 2.

Why Care?

I'll first make a list of programmers who I think have good reasons to **not** care.

- If you don't write code that's CPU or RAM bandwidth bound, you don't care. Majority of web developers fall into this category, the CPU-bound heavy lifting happens in external systems like TCP/IP stack or database engines.
- If you write code that does some math, but the actual math is implemented in external libraries. Many machine learning developers fall into this category, for the same reason they're OK using Python and similar high-level but slow languages for their job. Some C++ folks are happy with vectorized algorithms implemented in HPC libraries like Eigen and MKL and don't write performance-sensitive code of their own.
- If you write code that's not performance critical. Maybe a desktop app which idles most of the time waiting for user input. Maybe a cloud app where you have unlimited compute power as long as you have unlimited budget, and horizontal scaling is cheaper than optimizing code.
- If you use a language that doesn't have a good SIMD support. Languages like JavaScript or PHP usually don't, but SIMD is not limited to C++, there's solid support in modern .NET
- If you write numerical code which runs on GPGPUs, using CUDA, DirectCompute, or OpenCL.
- If you're happy with OpenCL. It has its limitations, and it has very different programming model close to that one of GPUs, but if you're OK with these limitations it might be easier to use than manual SIMD, while delivering comparable performance.

Applications

The obvious answer is "for code that multiplies long float32 vectors and inverts large matrices", but it's more than just that. Only the very first SSE 1 was limited to 32-bit floats. Personally, I've used it with great success in following areas.

1. For video games, and similar stuff like CAD/CAM/CAE. There's an awesome Microsoft's library DirectXMath which already implements many useful algorithms on top of SIMD. Despite the name, it's a general-purpose library, very useful for OpenGL and others. To use it efficiently, you need some understanding what it's all about. There's a third-party portable fork on github which compiles for Linux, OSX, iOS and Android, using either SSE or NEON SIMD depending on the target platform.
2. For code dealing with large volumes of voxels. Regardless whether you have floats, integers or bits in these voxels, well-implemented manual vectorization can speed up your code by an order of magnitude. For bits, a single `_mm_and_si128` instruction takes 0.33 to 0.25 CPU cycles to process 128 of them, you won't get anywhere close with scalar code.

3. For video processing. I once needed to convert video frames on CPU from RGB32 into NV12. I needed code to run fast, as I had a video stream of these frames, up to 60 FPS at 3840×2160 pixels / each. I've implemented and benchmarked a few versions, and the fastest one was SSE 2 code doing 16-bit fixed point math. That time I haven't tested AVX due to target platform limitation, I only did SSE 1, 2, 4.1, and FMA.
4. For image processing. Images typically have quite a few pixels.
5. For random stuff, both trivial and not. Here's an example of trivial one.

Hardware Support

The table below summarizes when a particular instruction set was introduced. The backward compatibility is good in CPUs. After something is introduced it usually stays, AMD's XOP and 3DNow are rare exceptions. In runtime you use CPUID at startup and check the bits it returns. This table is for design-time, when you're targeting PCs and deciding which of them you want to use.

	AMD	Intel
SSE 1	Athlon 64, 2003[3]	Pentium III, 1999[3]
SSE 2	Athlon 64, 2003	Pentium 4, 2000
SSE 3	Athlon 64 "Venice", 2004	Pentium 4 "Prescott", 2004
SSSE 3	Bobcat and Bulldozer, 2011	Penryn, 2007; Bonnell, 2008.
SSE 4.1	Jaguar, 2013; Bulldozer, 2011	Penryn, 2007; Silvermont, 2013.
FMA	Piledriver, 2012; not supported in Jaguar	Haswell, 2013
AVX 1	Jaguar, 2013; Bulldozer, 2011	Sandy Bridge, 2011
AVX 2	Excavator, 2015	Haswell, 2013 but only "Core i" and Xeon models, most Pentium and Celeron CPUs don't support that.

I think now in 2021 it's OK to require SSE up to and including SSE 4.1, and SSSE 3, and fail at startup if not supported.

About the rest of them (FMA, AVX 1, AVX 2), discretion is advised as the optimal tradeoff depends on the product. Videogames need compatibility, CAD/CAE software can specify hardware requirements, on servers you're pretty much guaranteed to have AVX 2. When you need compatibility but newer instructions bring too much profit, either ship separate version of your binaries, or do runtime dispatch. A good way to implement runtime dispatch, only check the support once at application startup, and cache C function pointers, or maybe a pointer to abstract C++ class.

Previous generation consoles (PlayStation 4, Xbox One) have AMD Jaguar inside. Current generation consoles (PlayStation 5, Xbox Series S and X) are using AMD Zen 2 which supports all of the above vector instruction sets.

Programming with SIMD

The main disadvantage of the technology is steep learning curve. It took me couple months to become somewhat familiar with the technology, and a year or so before I could say I'm comfortable using it. I had decades of prior experience programming classic C++, and years of prior experience programming GPUs. Didn't help. CPU SIMD is very different from both of them.

[3] Before SSE 1, both Intel and AMD had other vector stuff: MMX, 3DNow!, and some extended variants of both. These have been superseded by modern SIMD instruction sets discussed in this article. Technically, MMX instructions still run on modern CPUs from both vendors, despite 22 years old. Most of the integer subset of SSE 2 is a direct copy-paste of MMX, operating on registers twice as wide.

First it takes time to wrap the head around the concept. That all CPUs are actually vector ones, and every time you're writing `float a = b + c;` you're wasting 75% to 87.5% of CPU time, because it could have added 4 or 8 numbers with slightly different instruction, spending exactly same time.

Then it takes much more time to get familiar with actual instructions. When I started learning the technology, I can remember few times when I spent hours trying to do something, come up with a solution that barely outperformed scalar version, only to find out later there's that *one special instruction* which only takes a couple of them to do what I did in a page of code, improving performance by a factor of magnitude.

Another thing, the documentation is not great. I hope I have fixed it to some extent, but still, that official guide is only useful when you already know which intrinsics do you need. Most developers don't know what modern SIMD is capable of. I'm writing this article hoping to fix it.

Data Types

Most of the time you should be processing data in registers. Technically, many of these instructions can operate directly on memory, but it's rarely a good idea to do so. 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 integer, there're instructions to copy values from SIMD registers to general purpose ones.

In C++, the registers are exposed as variables of the 6 fundamental types. See the table on the right. In assembly code there's no difference between registers of the same size, these types are just for C++ type checking.

	16 bytes	32 bytes
32-bit float	`__m128`	`__m256`
64-bit float	`__m128d`	`__m256d`
Integers	`__m128i`	`__m256i`

The compiler assigns variables to registers automagically, but remember there're only 16 registers underneath the compiler. Or just 8 if you're building a 32-bit binary. If you define too many local variables, and write code with many dependencies preventing compiler from reusing registers, the compiler will evict some variables from registers to RAM on the stack. In some edge cases, this may ruin the performance. Profile your software, and review what the compiler did to your C++ code on the performance critical paths.

One caveat about these vector types, compilers define them as a structure or union with arrays inside. Technically you can access individual lanes of the vectors by using these arrays inside, the code compiles and works. The performance is not great, though. Compilers usually implement such code as a roundtrip from register to memory, and then back. Don't treat SIMD vectors that way, use proper shuffle, insert or extract intrinsics instead.

General Purpose Intrinsics

Casting Types

Assembly only knows about 2 vector data types, 16-bytes registers and 32-bytes ones. There're no separate set of 16/32 byte registers, 16-byte ones are lower halves of the corresponding 32-bytes.

But they are different in C++, and there're intrinsics to re-interpret them to different types. In the table below, rows correspond to source types, and columns represent destination type.

	__m128	__m128d	__m128i	__m256	__m256d	__m256i
__m128	=	_mm_castps_pd	_mm_castps_si128	_mm256_castps128_ps256		
__m128d	_mm_castpd_ps	=	_mm_castpd_si128		_mm256_castpd128_pd256	
__m128i	_mm_castsi128_ps	_mm_castsi128_pd	=			_mm256_castsi128_si256
__m256	_mm256_castps256_ps128			=	_mm256_castps_pd	_mm256_castps_si256
__m256d		_mm256_castpd256_pd128		_mm256_castpd_ps	=	_mm256_castpd_si256
__m256i			_mm256_castsi256_si128	_mm256_castsi256_ps	_mm256_castsi256_pd	=

All these intrinsics compile into no instructions, so they're practically free performance wise. They don't change bits in the registers, so 32-bit `1.0f` float becomes `0x3f800000` in 32-bit lanes of the destination integer register. When casting 16-byte values into 32 bytes, the upper half is undefined.

Converting Types

There're also instructions to do proper type conversion. They only support signed 32-bit integer lanes. *_mm_cvtepi32_ps* converts 4 integer 32-bit lanes into 32-bit floats, *_mm_cvtepi32_pd* converts first 2 integers into 64-bit floats, *_mm256_cvtpd_epi32* converts 4 doubles into 4 integers, setting higher 4 lanes of the output to zero. When converting from floats to integer, these instructions use rounding mode specified in MXCSR control and status register. See *_MM_SET_ROUNDING_MODE* macro for how to change that mode. That value is a part of thread state, so the OS persists the value across context switches[4].

There're other ones with extra 't' in the name which ignore MXCSR and always use truncation towards zero, like *_mm_cvttpd_epi32* and *_mm_cvttps_epi32*.

There're also instructions to convert floats between 32- and 64-bit.

Memory Access

There're many ways to load data from memory into registers.

- All types support regular loads, like *_mm_load_si128* or *_mm256_load_ps*. They require source address to be aligned by 16 or 32 bytes. They may crash otherwise but it's not guaranteed, depends on compiler settings.
- All types support unaligned loads, like *_mm_loadu_si128* or *_mm256_loadu_ps*. They're fine with unaligned pointers but aligned loads are faster.
- `__m128` and `__m128d` types support a few extra. They support single-lane loads, to load just the first lane and set the rest of them to 0.0, they are *_mm_load_ss* and *_mm_load_sd*. They support intrinsics to load them from memory in reverse order (requires aligned source address).

[4] That feature may cause hard to find numerical errors when SIMD is used in a thread pool environment, such as OpenMP. For this reason, instead of changing MXCSR I prefer SSE 4.1 rounding intrinsics.

- In AVX, `__m128`, `__m256` and `__m256d` have broadcast load instructions, to load single float or double value from RAM into all lanes of the destination variable. Also `__m256` and `__m256d` types have broadcast loads which read 16 bytes from RAM and duplicate the value in 2 halves of the destination register.
- AVX 1 introduced masked load instructions, they load some lanes and zero out others.
- AVX 2 introduced gather load instruction, like `_mm_i32gather_ps`, they take a base pointer, integer register with offsets, and also integer value to scale the offsets. Handy at times, unfortunately rather slow.
- Finally, in many cases you don't need to do anything special to load values. If your source data is aligned, you can just write code like `__m128i value = *pointer;` and it'll compile into an equivalent of regular load.

Similarly, there's many ways to store stuff: aligned, unaligned, single-lane, masked in AVX 1.

Couple general notes on RAM access.

1. On modern PCs, RAM delivers at least 8 bytes per load. If you have dual-channel RAM, it delivers 16 bytes. This makes it much faster to access memory in aligned 16- or 32-bytes blocks. If you look at the source code of a modern version of memcpy or memmove library routine, you'll usually find a manually vectorized version which uses SSE 2 instructions to copy these bytes around. Similarly, when you need something transposed, it's often much faster to load continuously in blocks of 16 or 32 bytes, then transpose in registers with shuffle instructions.
2. Many instructions support memory source for one of the operands. When compiling for SSE-only targets, this requires aligned loads. When compiling for AVX targets, this works for both aligned and unaligned loads. Code like `_mm_add_ps( v, _mm_loadu_ps( ptr ) )` only merges the load when compiling for AVX, while `_mm_add_ps( v, _mm_load_ps( ptr ) )` merges for all targets because the load is aligned (but might crash in runtime if the passed address is not actually a multiple of 16 bytes).
3. Most single-lane loads/store can only use lowest lane of destination/source register. If you're implementing a reduction like sum or average, try to do so you have result in the first lane. The exceptions are `_mm256_insertf128_ps`, `_mm256_insertf128_pd`, `_mm256_inserti128_si256` AVX instructions, they can load 16-byte vectors into higher half of the destination register with little to no performance penalty compared to regular 16-byte load.
4. There're instructions to load or store bypassing CPU caches, _mm*[256]*_stream_*something*. Can be useful for cases like video processing, when you know you're writing stuff that will only be loaded much later, after you've processed the complete frame, and probably by another CPU core (L1 and L2 caches are per-core, only L3 is shared).
5. Most current CPUs can do twice as many loads per cycle, compared to stores.

Initializing Vector Registers

Initializing with Zeros

All the vector types have intrinsics like `_mm_setzero_ps` and `_mm256_setzero_si256` to initialize a register with all zeros. It compiles into code like `xorps xmm0, xmm0, xmm0`. Zero initialization is very efficient because that XOR instruction has no dependencies, the CPU just renames a new register[5].

[5] The hardware has much more than 16, it's just 16 names in assembly. The hardware has thousands of them, the pipeline is deep and filled with in-flight instructions who need different versions of same logical registers.

Initializing with Values
CPUs can't initialize registers with constants apart from 0, but compilers pretend they can, and expose intrinsics like _mm*[256]*_set_*something* to initialize lanes with different values, and _mm*[256]*_set1_*something* to initialize all lanes with the same value. If the arguments are compile-time constants, they usually compile into read-only data in the compiled binary. Compilers emit the complete 16/32 bytes value, and they make sure it's aligned. if the arguments aren't known at compile time, the compilers will do something else reasonable, e.g. if the register is mostly 0 and you only setting one lane they'll probably emit _mm_insert_*something* instruction. If the values come from variables, the compiler may emit shuffles, or scalar stores followed by vector load.

For all their _mm*[256]*_set_*something* intrinsics, Intel screwed up the order. To make a register with integer values [1, 2, 3, 4], you have to write either `_mm_set_epi32( 4, 3, 2, 1 )` or `_mm_setr_epi32( 1, 2, 3, 4 )`.

Bitwise Instructions
Both floats and integers have a fairly complete set of bitwise instructions. All of them have AND, OR, XOR, and ANDNOT instructions. If you need bitwise NOT, the fastest way is probably XOR with all ones. Example for 16-byte values:

```
__m128i bitwiseNot( __m128i x )
{
	const __m128i zero = _mm_setzero_si128();
	const __m128i one = _mm_cmpeq_epi32( zero, zero );
	return _mm_xor_si128( x, one );
}
```

Whenever you need a register value with all bits set to 1, it's often a good idea to do what I did here, setzero then compare zeroes with zeroes.

Floating Point Instructions
Arithmetic
Most of them are available in all-lane versions, and single lane version which only compute single lane result, and copy the rest of the lanes from the first arguments.

Traditional
All 4 basic operations are implemented, add, subtract, multiply, divide. Square root instruction is also there, even for 64-bit floats.

Unorthodox
The CPU has minimum and maximum instructions. More often than not, modern compilers compile standard library functions `std::min<float>` / `std::max<double>` into `_mm_min_ss` / `_mm_max_sd`

For 32-bit floats, CPUs implement faster approximate versions of $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$ They are `_mm_rcp_ps` and `_mm_rsqrt_ps`, respectively[6]. For both of them the documentation says "maximum relative error for this approximation is less than 1.5*2^-12", translates to 3.6621×10^{-4} maximum relative error.

SSE 3 has horizontal add and subtract instructions, like `_mm_hadd_ps`, which takes two registers with [a, b, c, d] and [e, f, g, h] and returns [a+b, c+d, e+f, g+h]. The performance is not great, though.

[6] I use them very rarely. On old CPUs like Pentium 3, it took 56 CPU cycles to compute non-approximated square root, and 48 cycles to divide vectors. Faster approximations made a lot of sense back then. However, modern CPUs like Skylake or Ryzen do that in 12-14 cycles, little profit from these faster approximations.

SSE 3 also has a couple of instructions which alternatively add and subtract values. `_mm_addsub_ps` takes two registers with [a, b, c, d] and [e, f, g, h] and returns [a-e, b+f, c-g, d+h]. `_mm_addsub_pd` does similar thing for double lanes. Handy for multiplying complex numbers and other things.

SSE 4.1 includes dot product instruction, which take 2 vector registers and also 8-bit constant. It uses higher 4 bits of the constant to compute dot product of some lanes of the inputs, then lower 4 bits of the constant to broadcast the result.
For instance, when SSE 4.1 is enabled, `XMVector3Dot` library function compiles into single instruction, this one: `_mm_dp_ps( a, b, 0b01111111 )` The bits in the constant mean "compute dot product of the first 3 lanes ignoring what's in the highest ones, and broadcast the resulting scalar value into all 4 lanes of the output register". The lanes for which the store bit is zero will be set to `0.0f`.

SSE 4.1 has introduced rounding instructions for both sizes of floats. For 32-bit floats, the all-lanes version is exposed in C++ as `_mm_round_ps`, `_mm_ceil_ps`, and `_mm_floor_ps` intrinsics. For AVX, Intel neglected to implement proper ceil/round intrinsics, they only exposed `_mm256_round_ps` and `_mm256_round_pd` which take extra integer constant specifying rounding mode. Use `_MM_FROUND_NINT` constant to round to nearest integer[7], `_MM_FROUND_FLOOR` to round towards negative infinity, `_MM_FROUND_CEIL` to round towards positive infinity, or `_MM_FROUND_TRUNC` to round towards zero.

Missing

Unlike NEON, there's no SSE instructions for unary minus or absolute value. The fastest way is bitwise tricks, specifically `_mm_xor_ps( x, _mm_set1_ps( -0.0f ) )` for unary minus, `_mm_andnot_ps( _mm_set1_ps( -0.0f ), x )` for absolute value. The reason these tricks work, `-0.0f` float value only has the sign bit set, the rest of the bits are 0, so `_mm_xor_ps` flips the sign and `_mm_andnot_ps` clears the sign bit making the value non-negative.

There's no logarithm nor exponent, and no trigonometry either. Intel's documentation says otherwise, because Intel were writing their documentation for their C++ compiler, which implements them in its standard library. You can fall back to scalar code, or better search the web for the implementation. It's not terribly complex, e.g. trigonometric functions are usually implemented as high-degree minmax polynomial approximation. For single-precision floats you can use XMVectorSin/XMVectorCos/etc. from DirectXMath, for FP64 use the coefficients from GeometricTools.

Comparisons

There're all-lanes versions of equality comparison and the rest of them, <, >, ≤, ≥, ≠. These versions return another float register, that's either all zeros 0.0f, or all ones. A float with all ones is a NAN.

You can send the result from SIMD to a general-purpose CPU register with `_mm_movemask_ps`, `_mm_movemask_pd` or the corresponding AVX 1 equivalents. These instructions gather most significant bits of each float/double lane (that bit happen to be the sign bit, by the way), pack these bits into a scalar, and copy to a general-purpose CPU register. The following code prints 15:

```
const __m128 zero = _mm_setzero_ps();
const __m128 eq = _mm_cmpeq_ps( zero, zero );
const int mask = _mm_movemask_ps( eq );
printf( "%i\n", mask );
```

[7] More specifically, processors do bankers' rounding: when the fraction is exactly 0.5, the output will be nearest even number. This way 0.5 is rounded to 0, while 1.5 becomes 2.

The 0 == 0 comparison is true for all 4 lanes of `__m128`, the `eq` variable has all 128 bits set to 1, then `_mm_movemask_ps` gathers and returns sign bits of all 4 float lanes. `0b1111` becomes 15 in decimal.

Another thing you can do with comparison results, use them as arguments to bitwise instructions to combine lanes somehow. Or pass them into blendv_*something*. Or cast to integer vectors and do something else entirely.

Also, both 32- and 64-bit floats have instructions to compare just the lowest lanes of 2 registers, and return the comparison result as bits in the flags register accessible to general-purpose instructions:

```
// Compare the first lane for `x > 0.0f`
bool isFirstLanePositive( __m128 x )
{
	return (bool)_mm_comigt_ss( x, _mm_setzero_ps() );
}
```

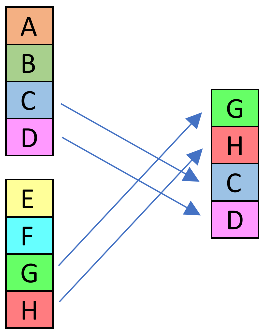
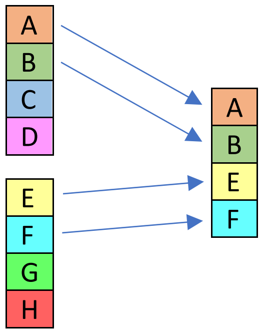
For AVX, Intel only specified two comparison intrinsics, `_mm256_cmp_ps` and `_mm256_cmp_pd`. To compare 32-byte vectors, you need to supply an integer constant for the comparison predicate, such as `_CMP_EQ_OQ` or other from that header. See this stackoverflow answer for more info on predicates.

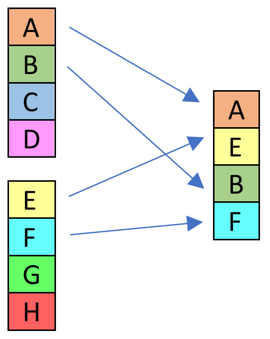
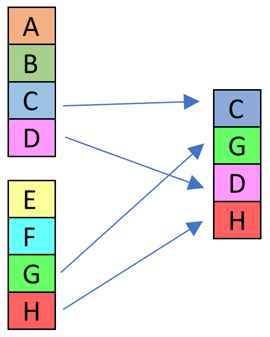
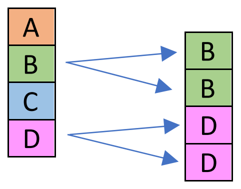
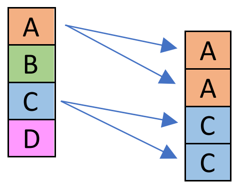
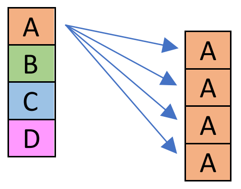
Shuffles

These are instructions which reorder lanes. They are probably the most complicated topic about the subject. Here's an example code which transposes a 4x4 matrix. To me it looks a bit scary, despite I understand pretty well what that code does.

We'll start with shuffle instructions processing 32-bit float lanes in 16-byte registers. On the images below, the left boxes represent what's on input, the right one what's on output. The A B C D values are lanes in these registers, the first one is on top.

Fixed Function Shuffles

_mm_movehl_ps	SSE 1	Move the upper 2 single-precision (32-bit) floating-point elements from "b" to the lower 2 elements of "dst", and copy the upper 2 elements from "a" to the upper 2 elements of "dst".	
_mm_movelh_ps	SSE 1	Move the lower 2 single-precision (32-bit) floating-point elements from "b" to the upper 2 elements of "dst", and copy the lower 2 elements from "a" to the lower 2 elements of "dst".	

_mm_unpacklo_ps	SSE 1	Unpack and interleave single-precision (32-bit) floating-point elements from the low half of "a" and "b", and store the results in "dst".	
_mm_unpackhi_ps	SSE 1	Unpack and interleave single-precision (32-bit) floating-point elements from the high half "a" and "b", and store the results in "dst".	
_mm_movehdup_ps	SSE 3	Duplicate odd-indexed single-precision (32-bit) floating-point elements from "a", and store the results in "dst".	
_mm_moveldup_ps	SSE 3	Duplicate even-indexed single-precision (32-bit) floating-point elements from "a", and store the results in "dst".	
_mm_broadcastss_ps	AVX 2	Broadcast the low single-precision (32-bit) floating-point element from "a" to all elements of "dst".	

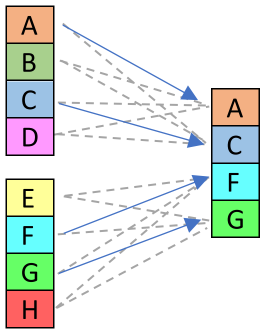
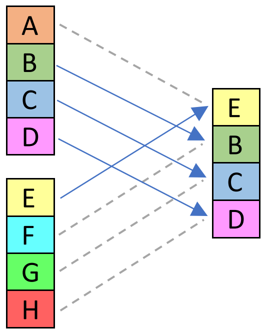
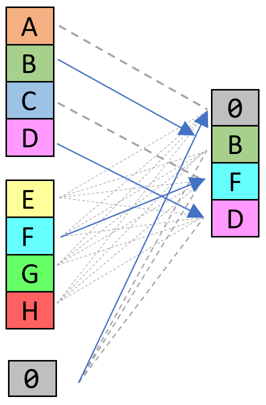
Variable Compile Time Shuffles

These instructions include the shuffle constant. This means you can't change that constant dynamically, it has to be a compile-time constant, like C++ `constexpr` or a template argument. The following C++ code fails to compile:

```
	const __m128 zero = _mm_setzero_ps();
	return _mm_shuffle_ps( zero, zero, rand() );
```

VC++ says "`error C2057: expected constant expression`"

In the pictures below, blue arrows show what has been selected by the control value used for the illustration. Dashed gray arrows show what could have been selected by different control constant.

_mm_shuffle_ps	SSE 1	Shuffle single-precision (32-bit) floating-point elements. On the image on the right, the control constant was `10 01 10 00 = 0x98`. The first 2 lanes come from the first argument, second 2 lanes from the second one. If you want to permute a single vector, pass it into both arguments.	
_mm_blend_ps	SSE 4.1	Blend packed single-precision (32-bit) floating-point elements from "a" and "b" using control mask. On the illustration on the right, the control was 1, it's `0b0001` in binary, so only the first lane was taken from the second argument.	
_mm_insert_ps	SSE 4.1	Insert single float lane, and optionally zero out some lanes. The image on the right shows what happens when the control value is `0b01100001` = 0x61: the source index is 1 that's why the F value is being inserted, the destination index is 2 so that value is inserted into lane #2, and the lowest 4 bits are `0001` so the output lane #0 is zeroed out. You can abuse this instruction to selectively zero out some lanes without inserting: pass same register in both arguments, and e.g. `0b00001001` control value to zero out lanes #0 and #3. The equivalent is *`_mm_blend_ps`* with *`_mm_setzero_ps`* second argument, but that's 2 instructions instead of one.	 A B C D E F G H 0 0 B F D
_mm_permute_ps	AVX 1	An equivalent of *`_mm_shuffle_ps`* with the same value in both arguments. Machine code of *`_mm_shuffle_ps`* is 1 byte shorter.	

I’m too lazy to draw similar block diagrams for `__m128d`, `__m256`, and `__m256d` shuffles. Fortunately, Intel was lazy as well. The 64-bit float versions are very similar to 32-bit floats. The shuffle constants have only half of the bits for 64-bit floats, the idea is the same.

As for the 32-byte AVX versions, the corresponding AVX instructions only shuffle/permute within 16-bit halves of the register. Here’s from documentation of *_mm256_shuffle_ps*: *Shuffle single-precision (32-bit) floating-point elements in "a"* ***within 128-bit lanes****.*

However, there’re also some new AVX2 shuffle instruction which can cross 16-byte lanes.

_mm256_broadcastss_ps broadcasts the lowest lane into all 8 destination ones. Same applies to *_mm256_broadcastsd_pd*, it broadcasts a 64-bit float into all laned of the output.

_mm256_permute2f128_ps, *_mm256_permute2f128_pd*, *_mm256_permute2x128_si256* shuffle 16-byte lanes from two 32-byte registers, it can also selectively zero out 16-byte lanes.

_mm256_permute4x64_pd and *_mm256_permute4x64_epi64* permute 8-byte lanes within 32-byte registers.

Variable Run Time Shuffles

SSE 4.1 introduced *_mm_blendv_ps* instruction. It takes 3 arguments, uses sign bit of the mask to select lanes from A or B.

No version of SSE has float shuffling instructions controllable in runtime. The closest one is *_mm_shuffle_epi8* from SSSE3, see the corresponding section under integer instructions. If you really need that, you can cast registers to integer type, use *_mm_shuffle_epi8*, then cast back[8].

AVX 1 finally introduced one, *_mm_permutevar_ps*. It takes 2 arguments, float register with source values, and integer vector register with source indices. It treats integer register as 32-bit lanes, uses lower 2 bits of each integer lane to select source value from the float input vector.

_mm256_permutevar8x32_ps from AVX 2 can move data across the complete 32-byte register.

Fused Multiply-Add

When processors vendors brag about how many FLOPs they can do per cycle, almost universally, they count FLOPs of their FMA instructions.

FMA instructions take 3 arguments, each being either 32- or 64-bit floating point numbers, a, b and c, and compute $(a \cdot b) + c$. For marketing reasons, that single operation counts as 2 FLOPs.

Modern CPUs implement 8-wide FMA for 32-bit floats, *_mm256_fmadd_ps* intrinsic in C++, and 4-wide FMA for 64-bit floats, *_mm256_fmadd_pd* intrinsics. There’re versions which subtract instead of adding, flip sign of the product before adding. There’re even versions which alternately add or subtract, computing $(a \cdot b) + c$ for even lanes $(a \cdot b) - c$ for odd ones, or doing the opposite. It was probably added for complex numbers multiplication.

There’re versions twice as narrow operating on 16-byte registers, 4-wide 32-bit floats FMA, and 2-wide 64-bit floats FMA.

These instructions process the same SIMD registers, `__m128`, `__m128d`, `__m256` or `__m256d` C++ types.

[8] But beware of bypass delays between integer and floating-point domains. On some old CPUs the latency can be up to 2 cycles. For more info, see section 13.6 “Using vector instructions with other types of data than they are intended for” of “Optimizing subroutines in assembly language” article.

Besides performance, FMA is also more precise. These instructions only round the value once, after they have computed both the product and the sum. The intermediate value has twice as many bits.

The support for these instructions is wide but not universal. Both Intel and AMD support the compatible version of FMA, called FMA 3, in their CPUs released since 2012-2013. See hardware support section for more info.

Another caveat, the latency of FMA is not great, 4-5 CPU cycles on modern CPUs. If you're computing dot product or similar, have an inner loop which updates the accumulator, the loop will throttle to 4-5 cycles per iteration due to data dependency chain. To resolve, unroll the loop by a small factor like 4, use 4 independent accumulators, and sum them after the loop. This way each iteration of the loop handles 4 vectors independently, and the code should saturate the throughput instead of stalling on latency. See this stackoverflow answer for the sample code which computes dot product of two FP32 vectors.

Integer Instructions

Arithmetic

Traditional

Additions and subtractions are totally fine and do what you would expect, e.g. `_mm_add_epi32` adds 32-bit lanes of two integer registers.

Moreover, for 8 and 16 bits, there're saturated versions of them, which don't overflow but instead stick to minimum or maximum values. For example, `_mm_adds_epi8( _mm_set1_epi8( 100 ), _mm_set1_epi8( 100 ) )` will return a vector with all 8-bit lanes set to +127, because the sum is 200 but maximum value for signed bytes is +127. Here's more in-depth article on saturation arithmetic.

There's no integer divide instruction. Intel has integer divide implemented in the standard library of their proprietary compiler. If you're dividing by a compile-time constant that's the same for all lanes, you can write a function which divides same sized scalar by that constant, compile with https://godbolt.org/ and port to SIMD. For example, here's what it compiled when asked to divide `uint16_t` scalar by 11, and here's how to translate that assembly into SIMD intrinsics:

```
// Divide uint16_t lanes by 11, all 8 of them in just two SSE 2 instructions.
__m128i div_by_11_epu16( __m128i x )
{
	x = _mm_mulhi_epu16( x, _mm_set1_epi16( (short)47663 ) );
	return _mm_srli_epi16( x, 3 );
}
```

Multiplication is tricky as well. `_mm_mul_epi32` takes 2 values [a,b,c,d] [e, f, g, h], ignores half of the values, and returns register with 2 64-bit values [a*e, c*g] The normal version is `_mm_mullo_epi32`. Oh, and both are SSE 4.1. If you only have SSE 2, you'll need workarounds.

Unorthodox

For most lane sizes there're integer minimum and maximum instructions, e.g. `_mm_max_epu8` (SSE 2) is maximum for uint8_t lanes, `_mm_min_epi32` (SSE 4.1) is minimum for 32-bit integer lanes.

SSSE 3 has absolute value instructions for signed integers. It also has horizontal addition and subtraction instructions, e.g. `_mm_hsub_epi32` takes [a, b, c, d] and [e, f, g, h] and returns [a-b, c-d, e-f, g-h]. For signed int16 lanes, saturated version of horizontal addition and subtraction is also available.

For uint8_t and uint16_t lanes, SSE 2 has instructions to compute average of two inputs.

The weirdest of them is probably *_mm_sad_epu8* (SSE2) / *_mm256_sad_epu8* (AVX2). The single instruction, which can run once per cycle on Intel Skylake, is an equivalent of the following code:

```
array<uint64_t, 4> avx2_sad_epu8( array<uint8_t, 32> a, array<uint8_t, 32> b )
{
        array<uint64_t, 4> result;
        for( int i = 0; i < 4; i++ )
        {
                uint16_t totalAbsDiff = 0;
                for( int j = 0; j < 8; j++ )
                {
                        const uint8_t va = a[ i * 8 + j ];
                        const uint8_t vb = b[ i * 8 + j ];
                        const int absDiff = abs( (int)va - (int)vb );
                        totalAbsDiff += (uint16_t)absDiff;
                }
                result[ i ] = totalAbsDiff;
        }
        return result;
}
```

I think it was added for video encoders. Apparently, they treat AVX 2 register as a block of 8×4 grayscale 8-bit pixels, and want to estimate compression errors. That instruction computes sum of errors over each row of the block. I've used it couple times for things unrelated to video codecs, it's the fastest ways to compute sum of bytes: use zero vector for the second argument of *_mm_sad_epu8*, then *_mm_add_epi64* to accumulate.

Comparisons

Only all-lane versions are implemented. The results are similar to float comparisons, they set complete lane to all zeroes or all ones. For example, _mm_cmpgt_epi8 sets 8-bit lanes of the output to either 0 or 0xFF, depending on whether the corresponding signed 8-bit values are greater or not.

Protip: a signed integer with all bits set is equal to -1. A useful pattern to count matching numbers is integer subtract after the comparison:

```
const __m128i cmp = _mm_cmpgt_epi32( val, threshold );     // Compare val > threshold
acc = _mm_sub_epi32( acc, cmp );     // Increment accumulator where comparison was true
```

Beware of integer overflows though. Rarely important for counting 32-bit matches (they only overflow after 4G of vectors = 64GB of data when using SSE2), but critical for 8 and 16-bit ones. One approach to deal with that, a nested loop where the inner one consumes small batch of values which guarantees to not overflow the accumulators, and outer loop which up-casts to wider integer lanes.

There're no unsigned integer comparison instructions. If you need them, here's an example how to implement manually, on top of signed integer comparison.

```
// Compare uint16_t lanes for a > b
__m128i cmpgt_epu16( __m128i a, __m128i b )
{
        const __m128i highBit = _mm_set1_epi16( (short)0x8000 );
        a = _mm_xor_si128( a, highBit );
        b = _mm_xor_si128( b, highBit );
        return _mm_cmpgt_epi16( a, b );
}
```

To compare integer lanes for `a <= b`, a good way (two fast instructions) is this: `min( a, b ) == a`.

The move mask instruction is only implemented for 8-bit lanes. If you want results of 32-bit integer comparison in a general-purpose register, 1 bit per lane, one workaround is cast `__m128i` into `__m128` and use *_mm_movemask_ps*. Similar for 64-bit lanes, cast into `__m128d` and use *_mm_movemask_pd*.

Shifts

Whole Register Shift

It can only shift by whole number of bytes.

The whole register shifts are `_mm_srli_si128` and `_mm_slli_si128` in SSE 2. AVX 2 equivalents like `_mm256_slli_si256` independently shift 16-byte lanes, so the bytes that would have been shifted across 16-byte halves of the register become zeros. If you want to shift the entire 32-byte AVX register, you gonna need a workaround, see this stackoverflow answer.

SSSE 3 has `_mm_alignr_epi8` instruction. Takes 16 bytes in one register, 16 bytes in another one, concatenates them into 32-bit temporary result, shifts that one right by whole number of bytes, and keep the lowest 16 bytes.

Individual Lanes Shift

SSE 2 has instructions which encode shift amount in the opcodes, and versions which take it from the lowest lane of another vector register. The shift amount is the count of bits to shift, it's applied to all lanes. Right shift instructions come in two versions, one shifting in zero bits, another shifting in sign bits. `_mm_srli_epi16( x, 4 )` will transform 0x8015 value into 0x0801, while `_mm_srai_epi16( x, 4 )` will transform 0x8015 into 0xF801. They likely did it due to the lack of integer division: `_mm_srai_epi16( x, 4 )` is an equivalent of `x/16` for signed `int16_t` lanes.

Variable Shifts

AVX2 introduced instructions which shift each lane by different amount taken from another vector. The intrinsics are `_mm_sllv_epi32`, `_mm_srlv_epi32`, `_mm_sllv_epi64`, `_mm_srlv_epi64`, and corresponding 32-bytes versions with `_mm256_` prefix. Here's an example which also uses the rest of the integer shifts.

Pack and Unpack

Unlike floats, the same data types, `__m128i` and `__m256i`, can contain arbitrary count of lanes. Different instructions view them as 8-, 16-, 32-, or 64-bit lanes, either signed or unsigned. There're many instructions to pack and unpack these lanes.

Unpack instructions come in 2 versions. unpacklo_*something* unpacks and interleaves values from the low half of the two registers. For example, `_mm_unpacklo_epi32` takes 2 values, [a,b,c,d] and [e,f,g,h], and returns [a,e,b,f]; `_mm_unpackhi_epi32` takes 2 values, [a,b,c,d] and [e,f,g,h], and returns [c,g,d,h]. One obvious application, if you supply zero for the second argument, these instructions will convert unsigned integer lanes to wider ones, e.g. 8-bit lanes into 16-bit ones, with twice as few lanes per register.

The opposite instructions, for packing lanes, come in 2 versions, for signed and unsigned integers. All of them use saturation when packing the values. If you don't want the saturation, bitwise AND with a constant like `_mm_set1_epi16( 0xFF )` or `_mm_set1_epi32( 0xFFFF )`[9], and unsigned saturation won't engage.

`_mm_packs_epi16` takes 2 registers, assumes they contain 16-bit signed integer lanes, packs each lane into 8-bit signed integer using saturation (values that are greater than +127 are clipped to +127, values that are less than -128 are clipped to -128), and returns a value with all 16 values.

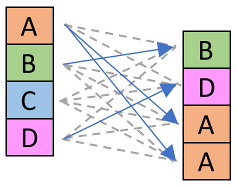
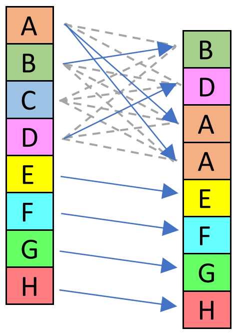
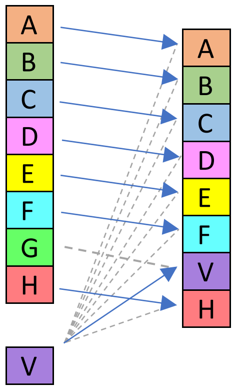
`_mm_packus_epi16` does the same but it assumes the input data contains 16-bit unsigned integer lanes, that one packs each lane into 8-bit unsigned integer using saturation (values that are greater than 255 are clipped to 255), and returns a value with all 16 values.

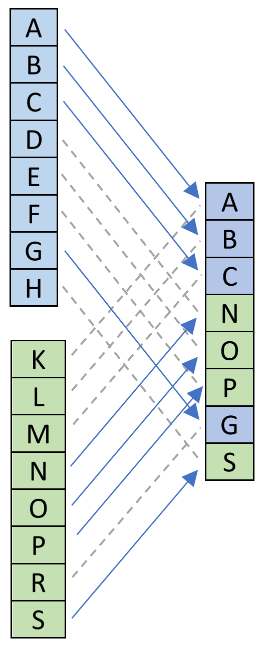
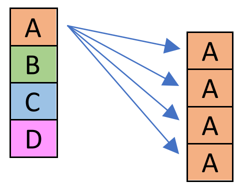
[9] Or you can build the magic numbers in code. It takes 3 cycles, setzero, cmpeq_epi32, unpacklo_epi8/ unpacklo_epi16. In some cases, can be measurably faster than a RAM load to fetch the constant.

Before AVX2, unpacking with zeroes was the best way to convert unsigned integers to wider size. AVX2 introduced proper instructions for that, e.g. `_mm256_cvtepu8_epi32` takes 8 lowest bytes of the source, and converts these bytes into a vector with 8 32-bit numbers. These instructions can even load source data directly from memory.

Shuffles

Besides pack and unpack, there're instructions to shuffle integer lanes.

_mm_shuffle_epi32	SSE 2	Shuffle 32-bit integer lanes. The picture on the right is for the shuffle constant `0b00001101` which is `0x0D` in hexadecimal.	
_mm_shufflelo_epi16	SSE 2	Shuffles lower 4 of the 16-bit integer lanes. The upper 64 bit is copied from source do destination. The picture on the right is for shuffle constant `0x0D`.	 A B C D E F G H B D A A E F G H
_mm_shufflehi_epi16	SSE 2	Similar to the above, copies the lower 4 lanes, and shuffles the higher 4 lanes into the higher lanes of the result.	
_mm_insert_epi16	SSE 2	Insert a 16-bit integer value. Unlike the floating-point inserts, the integer to insert V comes from a general-purpose register. If you have it in another vector register, you may want to do something else, like a shift followed by blend. The index where to insert is encoded into the instruction. On the picture to the right, that index was 6.	 A B C D E F G H V A B C D E F V H
_mm_insert_epi8, _mm_insert_epi32, _mm_insert_epi64	SSE 4.1	Similar to the above but inserts 8-, 32- or 64-bit lanes.	
_mm_shuffle_epi8	SSSE 3	Shuffle 8-bit lanes, taking shuffle indices from another vector register. See the separate section of this article for more information.	

_mm_blend_epi16	SSE 4.1	Blend 16-bit lanes. The blend bit mask is encoded into the instruction. On the illustration to the right, that constant was `0b10111000 = 0xB8`	 A B C D E F G H K L M N O P R S A B C N O P G S
_mm_blend_epi32	AVX 2	Similar to the above, but blends 32-bit lanes instead of 16-bit ones, and marginally faster.	
_mm_blendv_epi8	SSE 4.1	Blends 8-bit lanes, but unlike the rest of them, blending bit mask is not encoded in the instruction, it comes from another, third input register. The instruction uses highest bit of each 8-bit lane to select each lane of the result from either of the two first input registers.	
_mm256_ permutevar8x32_epi32	AVX 2	A rare instruction which can permute values across 128-bit lanes, and the permite constants are not encoded in the instruction.	
_mm_broadcastb_epi8, _mm_broadcastw_epi16, _mm_broadcastd_epi32, _mm_broadcastq_epi64	AVX 2	Broadcast the lowest lane to the rest of them. The picture to the right is for _mm_broadcastd_epi32 instruction.	

_mm_shuffle_epi8

This instruction is a part of SSSE 3 set, and it deserves a dedicated section in this article. It's the only SSE instruction that implements runtime-variable shuffles. Unlike the rest of the shuffles, this one takes shuffle values from a vector register. Here's a C++ equivalent of that instruction:

```cpp
array<uint8_t, 16> shuffle_epi8( array<uint8_t, 16> a, array<uint8_t, 16> b )
{
	array<uint8_t, 16> result;
	for( int i = 0; i < 16; i++ )
	{
		const uint8_t mask = b[ i ];
		if( 0 != ( mask & 0x80 ) )
			result[ i ] = 0;
		else
			result[ i ] = a[ mask & 0xF ];
	}
	return result;
}
```

Only unlike the above C++ code with the loop, _mm_shuffle_epi8 has 1 cycle latency, and the throughput is even better, the computer I'm using to write this article can run 2 or them per clock.

Can be quite useful in practice. Here's one example which uses the first argument as a lookup table, and the second one as indices to lookup, to compute Hamming weight of integers. In some cases, that version can be faster than dedicated POPCNT instruction implemented in modern CPUs for counting these bits. Here's another, this one uses it for the intended purpose, to move bytes around.

The AVX 2 equivalent, *_mm256_shuffle_epi8*, shuffles within 128-bit lanes, applying the same algorithm onto both 128-bit pieces, using the corresponding piece of the second argument for shuffle control values.

Miscellaneous Vector Instructions

There're intrinsics to copy the lowest lane from vector register into general purpose one, such as *_mm_cvtsi128_si32*, *_mm_cvtsi128_si64*. And the other way too, *_mm_cvtsi32_si128*, *_mm_cvtsi64x_si128*, they zero out the unused higher lanes.

Similar ones are available for both floats types, *_mm_cvtss_f32*, but only 1 way, from vector register to normal C++ floats. For the other way, to convert scalar floats into SIMD registers, I usually use *_mm_set_ps* or *_mm_set1_ps*. Scalar floats are often already in SIMD registers, compilers need to do very little work, maybe nothing at all, maybe a shuffle.

_mm_extract_epi16 (SSE 2) extracts any of the 16-bit lanes (index encoded in the instruction) and returns the value in general purpose register. *_mm_extract_epi8*, *_mm_extract_epi32*, *_mm_extract_epi64*, *_mm_extract_ps* (all SSE 4.1) do the same for other lane sizes. The interesting quirk about *_mm_extract_ps*, it can return 32-bit float in a general-purpose integer register like `eax`.

Another useful SSE 4.1 instruction is `ptest`. It takes 2 vector registers, computes bitwise `( a & b )` and sets one flag if each and every bit of the result is 0. It also computes `( ( ~a ) & b )`, and sets another flag if that value is exactly 0 on all bits. These flags are readable by general-purpose CPU instructions. It's exposed in C++ as multiple intrinsics, *_mm_testz_si128*, *_mm_testnzc_si128*, *_mm_test_all_zeros*, *_mm_test_all_ones*, etc.

SSE 3 has a few unusual integer instructions. *_mm_mulhrs_epi16* does something like this to signed int16_t lanes: `return (int16_t)( ( (int)a * (int)b + 0x4000 ) >> 15 )` I have used it to apply volume to 16-bit PCM audio. _mm_sign_*something* multiplies signed lanes by signs of corresponding lane in the second argument, either -1, 0 or +1. *_mm_maddubs_epi16* does an equivalent of the following C++ code:

```
array<int16_t, 8> mad_bs( array<uint8_t, 16> a, array<int8_t, 16> b )
{
        array<int16_t, 8> result;
        for( int i = 0; i < 16; i += 2 )
        {
                const int p1 = (int16_t)a[ i ] * (int16_t)b[ i ];
                const int p2 = (int16_t)a[ i + 1 ] * (int16_t)b[ i + 1 ];
                int sum = p1 + p2;
                sum = std::max( sum, -32768 );
                sum = std::min( sum, +32767 );
                result[ i / 2 ] = (int16_t)sum;
        }
        return result;
}
```

Besides integer and floating-point math, modern CPUs can do other things with these vector registers. Hardware AES and SHA cryptographic algorithms are implemented on top of them. Intel implemented some strange string-handling instructions in SSE 4.2. This article covers general-purpose programming with focus on numerical computations, they aren't covered here.

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 100-300 cycles. Caches are faster than that but still slow, L3 cache takes 40-50 cycles to load, L2 cache around 10 cycles, even L1 cache is noticeably slower to access than registers. Keep your data structures SIMD-friendly. Prefer `std::vector` or equivalents like `CAtlArray` or `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. If you have to traverse a linked list or a graph while computing something for each node, sometimes `_mm_prefetch` intrinsic helps.

For optimal performance, RAM access needs to be aligned[10]. If you have `std::vector<__m128>` the standard library should align automatically, but sometimes you want aligned vectors of floats or types like `DirectX::SimpleMath::Vector3` which don't have sufficient natural alignment. For these cases, you can use custom allocator, tested on Windows and Linux.

When you're dealing with pairs of 32-bit float numbers (like FP32 vectors in 2D), you can load/store both scalars with a single instruction intended for FP64 numbers. You only need type casting for the pointer, and `_mm_castps_pd` / `_mm_castpd_ps` intrinsics for casting vectors. Similarly, you can abuse FP64 shuffles/broadcasts to move pairs of FP32 values in these vectors. Old CPUs have latency penalty for passing vectors between integer and float parts of CPU cores, but that's irrelevant because FP32 and FP64 are both floats.

There're good vectorized libraries for C++: Eigen, DirectXMath, couple of others. Learn what they can do and don't reinvent wheels. They have quite complex stuff already implemented there. If you enjoy looking at scary things, look at the source code of SSE version of XMVector3TransformCoordStream library routine.

When you're implementing complex SIMD algorithms, sometimes it's a good idea to create C++ classes. If you have a class with a couple of `__m128` fields, create it on the stack, and never create references nor pointers to it, VC++ compiler normally puts 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 often applies to small `std::array`-s of SIMD vectors, or C arrays.

Don't write `static const __m128 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 `LoadLibrary` of your DLL returns. Or you can place that value in a local non-static const variable.

`<xmmintrin.h>` library header has a macro to make shuffle constants for `_mm_shuffle_ps` and `_mm_shuffle_epi32`, `_MM_SHUFFLE`.

[10] More specifically, the memory access should not cross cache line boundary. Cache line size is 64 bytes and they are aligned by 64. When SIMD vectors are aligned properly (by 16 bytes for SSE, or by 32 bytes for AVX vectors), a memory access is guaranteed to only touch a single cache line.

When compiling for 32-bit, all the general-purpose registers are 32-bit as well. SIMD intrinsics which move 64-bit integers between SIMD registers and general-purpose ones are not available on the platform. To work around, use load/store, or fetch two 32-bit values with separate instructions.

If you use VC++, spam `__forceinline` on performance-critical SIMD functions which are called from hot loops. Code often contains magic numbers, also variables which don't change across iterations. Unlike scalar code, SIMD magic numbers often come from memory, not from the instruction stream. When compiler is told to `__forceinline`, it can load these SIMD constants once, and keep them in vector registers across iterations. Unless they are evicted to RAM due to registers shortage. Without inlining, the code will be reloading these constants in consuming functions. I've observed 20% performance improvement after adding `__forceinline` keywords everywhere. Apparently, VC++ inlining heuristics are tuned for scalar code, and fail for code with SIMD intrinsics.

Unfortunately, this means you probably can't use C++ lambdas on hot paths, as there's no way to mark lambda's `operator()` with `__forceinline`. You'll have to write custom classes instead, class methods and operators support `__forceinline` just fine.

If you're using gcc or clang to compile your code, they're better with inlining but forcing may still help sometimes, you can define `__forceinline` as a macro:

```
#define __forceinline inline __attribute__((always_inline))
```

If you're implementing dynamic dispatch to switch implementation of some vector routines based on supported instruction set, apply `__vectorcall` convention to the function pointers or virtual class methods. Such functions pass arguments and return value in vector registers. Can be measurable difference for 32-bit binaries. The default 64-bit convention ain't that bad, you probably won't get much profit for your 64-bit builds.

Agner Fog has resources for SIMD programming on his web site. The "Optimizing subroutines in assembly language" is extremely useful, also timings tables. He also has a vector class library with Apache license. I don't like wrapper classes approach: sometimes compilers emit suboptimal code due to these classes, many available instructions are missing from the wrappers, vectorized integer math often treats vectors as having different lanes count on every line of code so you'll have to cast them a lot. But parts like floating point exponent, logarithm and trigonometry are good.

Speaking of timing tables, this web site is an awesome source of performance-related data for individual instructions: https://www.uops.info/table.html