intel-intrinsics

Not intrinsically about intrinsics

By Guillaume Piolat

intel-intrinsics

Please use my library

By Guillaume Piolat

This is a talk about performance

Part 1
Speed is still important

Part 2
The D SIMD landscape

Part 3
How `intel-intrinsics` was made

Part 4
Choosen examples

Part 5
I'll tell you to profile your code first

Hello

- Auburn Sounds is a bootstrapped B2C music app business

- Clients = mostly urban music producers

- Complexity = about 80 kloc of D
- Open Source core = Dplug

- Competition is 99% C++

Selling audio plug-ins

100x real-time

- Audio plug-ins = small dynlibs that process audio quicker than real-time
- Fierce competition
- **CPU time is shared** (~1%)
- Typical commercial plug-in is between 10x to 300x real-time

Performance an enabler

- Rarely mentionned by B2C consumers *as long as software is fast enough*

- Many Quality vs CPU trade-offs
  Speed **enables** better-sounding algorithms

- Audio not special

Performance an enabler

- Rarely mentionned by B2C consumers *as long as software is fast enough*
- Many Quality vs CPU trade-offs
  Speed **enables** better-sounding algorithms
- Audio not special

YOUR CUSTOMERS PROBABLY LOVE PERFORMANCE

EVEN IF THEY DON'T TELL YOU

How to get faster programs?

- Measure, have a baseline, improve precision
  (cf. Alexandrescu talks)
- Make identified bottlenecks faster

How to get faster programs?

- Measure, have a baseline, improve precision
  (cf. Alexandrescu talks)
- Make identified bottlenecks faster

Single Instruction, Multiple Data helps.

But which D SIMD facility to use?

The D SIMD Landscape

(this image generated with goart.fotor.com)

Option #1: inline assembly

```
asm nothrow @nogc
{
    movd XMM0, A;
    movd XMM1, B;
    movd XMM2, C;
    movd XMM3, D;
    pxor XMM4, XMM4;

    punpcklbw XMM0, XMM4;
    punpcklbw XMM1, XMM4;
    punpcklbw XMM2, XMM4;
    punpcklbw XMM3, XMM4;

    punpcklwd XMM0, XMM4;
    punpcklwd XMM1, XMM4;
    punpcklwd XMM2, XMM4;
    punpcklwd XMM3, XMM4;

    cvtdq2ps XMM0, XMM0;
    cvtdq2ps XMM1, XMM1;
    cvtdq2ps XMM2, XMM2;
    cvtdq2ps XMM3, XMM3;

    movss XMM4, fxm1;
    pshufd XMM4, XMM4, 0;
    movss XMM5, fx;
    pshufd XMM5, XMM5, 0;
    mulps XMM0, XMM4;
    mulps XMM1, XMM5;
    mulps XMM2, XMM4;
    mulps XMM3, XMM5;
    movss XMM4, fym1;
    pshufd XMM4, XMM4, 0;
    movss XMM5, fy;
    pshufd XMM5, XMM5, 0;
    addps XMM0, XMM1;
    addps XMM2, XMM3;
    mulps XMM0, XMM4;
    mulps XMM2, XMM5;
    addps XMM0, XMM2;
    movups asmResult, XMM0;
}
```

Sample from Dplug, linear texture sampling

Option #1: using assembly

PROS

- Portable across DMD and LDC
- Predictable
- Debug performance

CONS

- Write twice, for x86 and x86_64 (except rare cases)
- Hard to write, debug, and read
- Very arch-specific

Option #1: using assembly

PROS

- Portable across DMD and LDC
- Predictable
- Debug performance

CONS

- Write twice, for x86 and x86_64 (except rare cases)
- Hard to write, debug, and read
- Very arch-specific
- Rarely the best performance
- Does not get faster over time

Option #2: `core.simd`

```d
void main()
{
    float4 A = [1.0f, 2, 3, 4];

    // access to elements
    float C = A.array[1];
    A.array[0] = C;
    assert(A.array[0] == 2);

    // vector ops
    int4 v = 7;
    v = 3 + v;
}
```

Introduced in 2012.

Option #2: `core.simd`

PROS

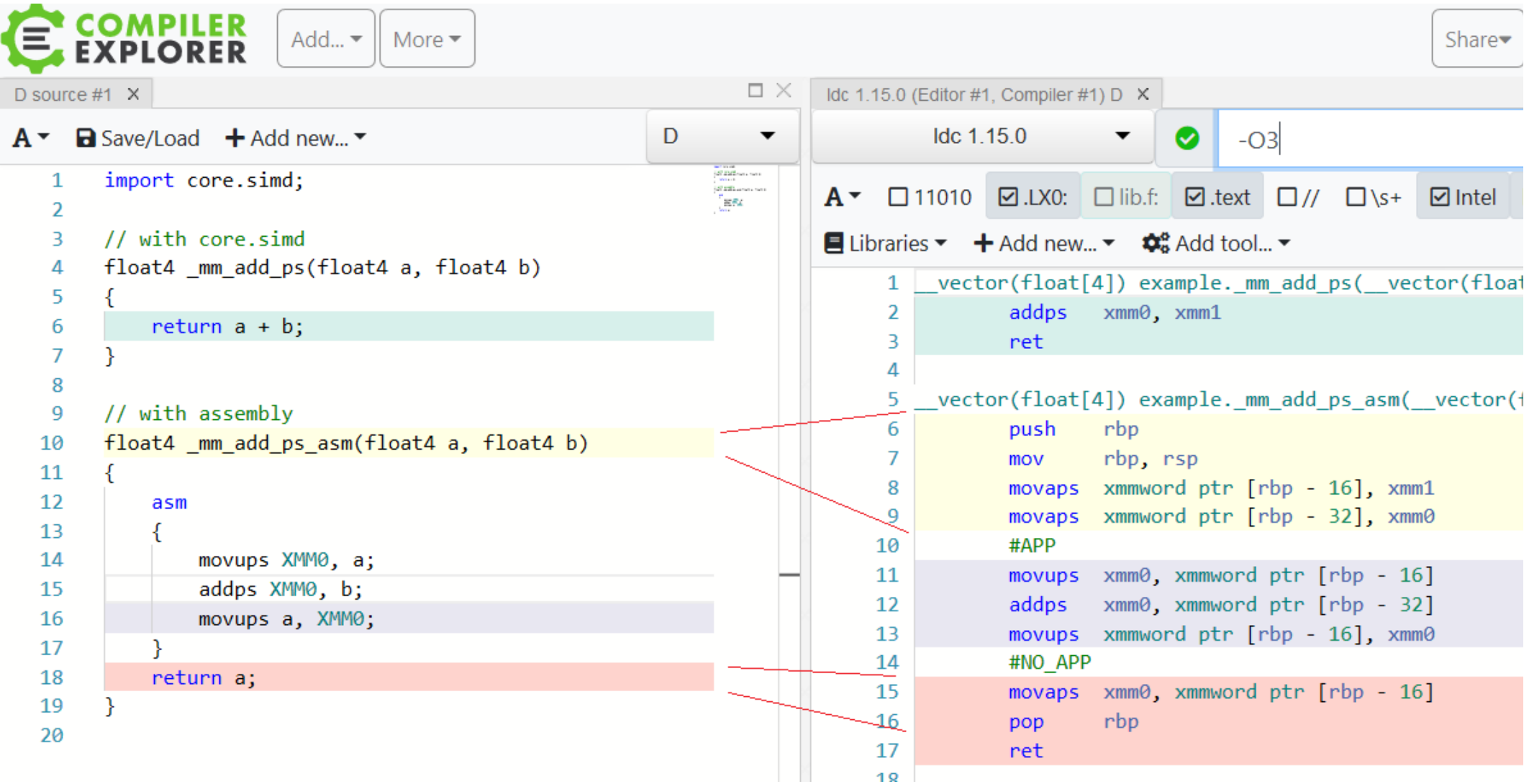
- Portable across DMD, LDC and GDC
- Easy to read/write/debug
- Pleasant syntax

CONS

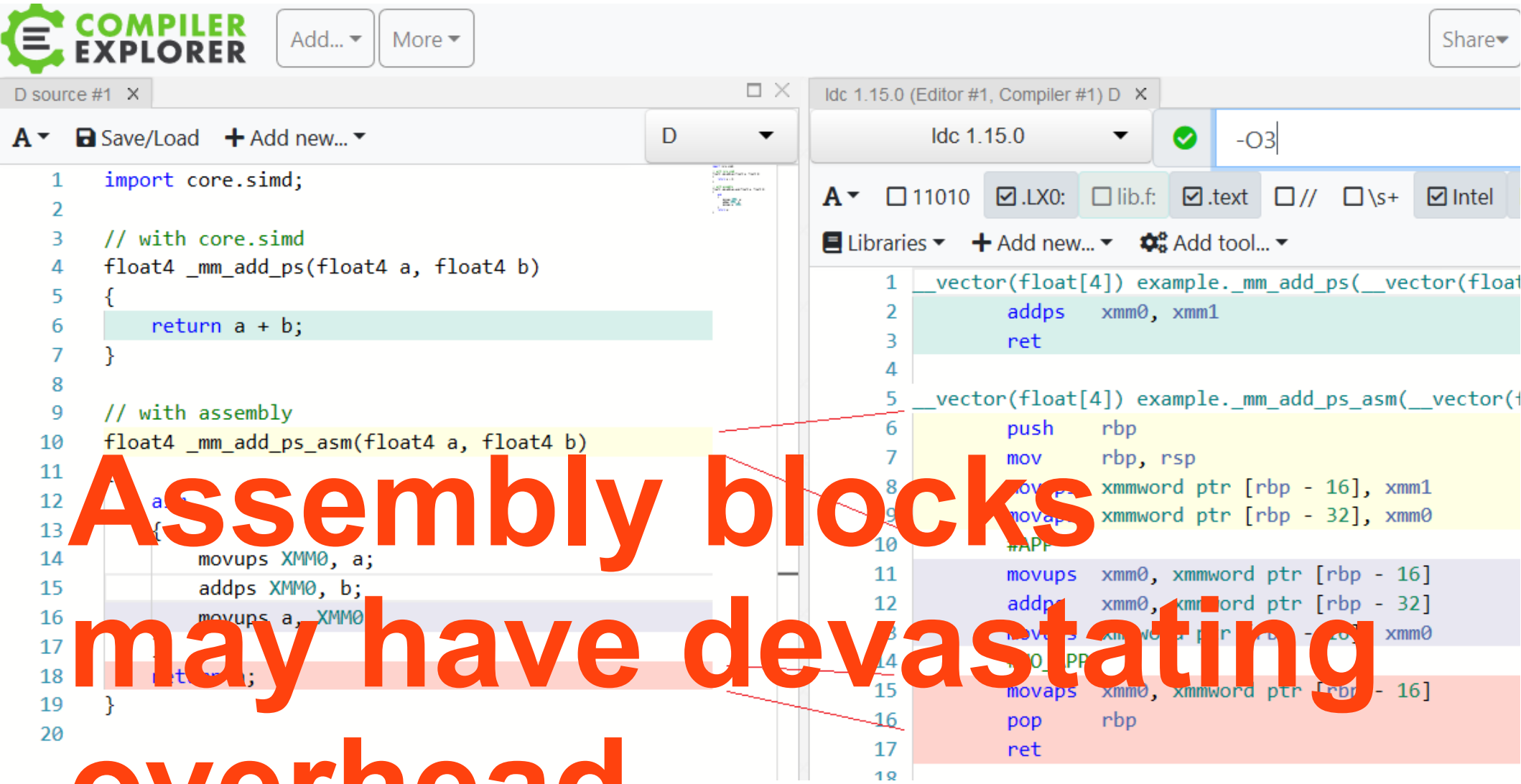
- No support in DMD + Win32
- x86 CPU have more operations than that

  eg :
  PMADDW
  PSHUFB...

Working with the back-end



Working with the back-end



Assembly blocks may have devastating overhead

Option #2: `core.simd`

PROS

- Portable across DMD, LDC and GDC
- Easy to read/write/debug
- Pleasant syntax

CONS

- No support in DMD + Win32
- x86 CPU have more operations than that

  eg :
  PMADDW
  PSHUFB...

`core.simd` is great

Option #3: `core.simd` + `D_SIMD`

```d
import core.simd;

void main()
{
    version(D_SIMD)
    {
        float4 a;
        a = simd!(XMM.PXOR)(a, a);
    }
}
```

A DMD extension also introduced in 2012.

Option #3: `core.simd` + `D_SIMD`

PROS

- Good x86 instruction set support

CONS

- `D_SIMD` only in DMD
- again, not in Win32

Option #4: `ldc.simd`

```
import core.simd, ldc.simd;

float4 test(float* ptr)
{
    return loadUnaligned!float4(ptr);
}
```

Extends `core.simd` with portable operations:

- `shufflevector`
- Unaligned load/store
- and more...

Some of it made it back to `core.simd`

Option #4: `ldc.simd`

PROS

- All the pros from `core.simd`
- Portable

CONS

- LDC-specific
- Many x86 operations not doable:

  eg: ADDSS,
  PMADDW,
  PAVGB...

Option #4: `ldc.simd`

PROS

- All the pros from `core.simd`
- Portable

CONS

- LDC-specific
- Many x86 operations not doable:

  eg: ADDSS,
  PMADDW,
  PAVGB...

Tension right here

Option #5: `ldc.gccbuiltins_x86`

```d
import core.simd;
import ldc.gccbuiltins_x86;

void testSIMD()
{
    float4 A = [1.0f, 2, 3, 4];
    A = __builtin_ia32_rsqrtss(A);
}
```

Extends `core.simd` with some x86 builtins

Option #5: `ldc.gccbuiltins_x86`

PROS

- Provide direct instruction generation.

CONS

- LDC only

Option #5: `ldc.gccbuiltins_x86`

PROS

- Provide direct instruction generation.

CONS

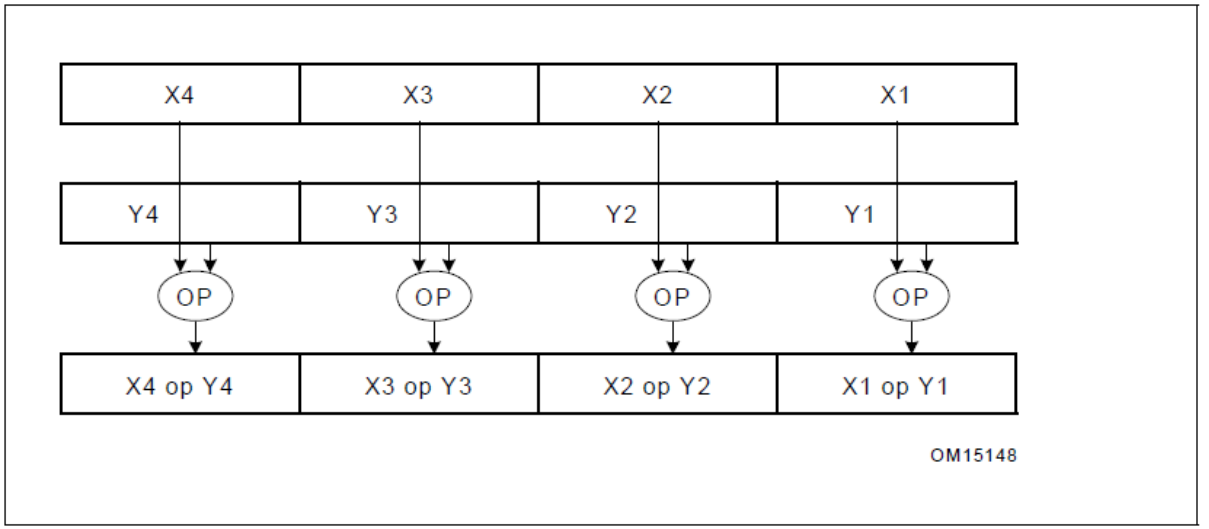
- LDC only

`intel-intrinsics`
started as a familiar syntax for
`ldc.gccbuiltins_x86`

How `intel-intrinsics` was made

Implementing _mm_add_ps

ADDPS instruction

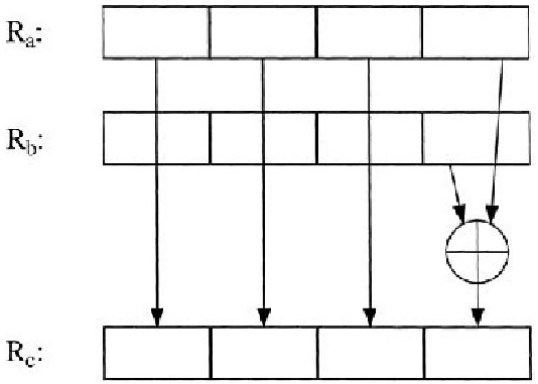


With `core.simd`:

```
alias __m128 = float4;

__m128 _mm_add_ps(__m128 a, __m128 b) pure @safe
{
    return a + b;
}
```

Implementing _mm_add_ss

ADDSS instruction



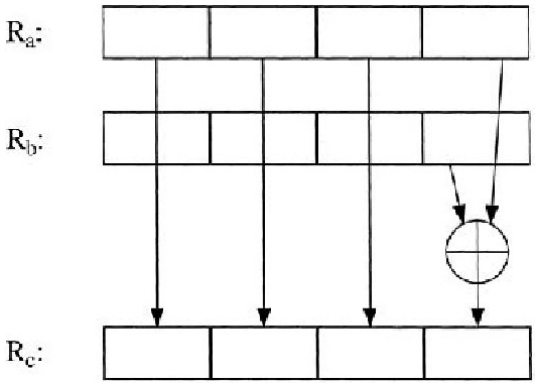

```
import ldc.gccbuiltins_x86;
alias _mm_add_ss = __builtin_ia32_addss;
```

With `ldc.gccbuiltins_x86`

```
4081  pragma(LDC_intrinsic, "llvm.x86.sse.add.ss")
4082     float4 __builtin_ia32_addss(float4, float4) pure @safe;
```

LDC 1.1 removed `__builtin_ia32_addss`!

ADDSS instruction




```d
import ldc.gccbuiltins_x86;
alias _mm_add_ss = __builtin_ia32_addss;
```

With `ldc.gccbuiltins_x86`

```d
4081  pragma(LDC_intrinsic, "llvm.x86.sse.add.ss")
4082      float4 __builtin_ia32_addss(float4, float4) pure @safe;
```

Option #5: `ldc.gccbuiltins_x86`

PROS

- Provide direct instruction generation.

CONS

- LDC only
- The built-ins are disappearing over time

LDC 1.1 removed `__builtin_ia32_addss!`

p0nce commented on 1 Mar 2017 • edited

These intrinsics have disappeared:

```
__builtin_ia32_mulss
__builtin_ia32_divss
__builtin_ia32_addss
__builtin_ia32_pmaxsw128
__builtin_ia32_pmaxub128
__builtin_ia32_pminsw128
__builtin_ia32_pminub12'
__builtin_ia32_pshufd
__builtin_ia32_pshufhw
__builtin_ia32_pshuflw
__builtin_ia32_storelv4si
__builtin_ia32_storedqu
__builtin_ia32_storeupd
```

were in LDC 1.0 but not 1.1.

I guess there is another way to do it with SIMD vector extensions?

LDC issues #2019, #2250 and #2759

What « intrinsics »?

kinke commented on 1 Mar 2017 • edited ▾ Member

No idea where those 'intrinsics' came from, afaik LDC never supported these directly. Where were they declared?

Of course there's other ways to deal with SIMD; `ldc.simd` and `ldc.llvmasm` provide ways to insert textual LLVM IR and/or inline assembly (besides more generic helpers, e.g., for shuffling), giving you tremendous flexibility - via a tedious interface. ;)

Are you sure your optimizations on this low level actually pay off, i.e., does the LLVM optimizer/vectorizer not produce sufficiently efficient code with appropriate command-line options?

What « intrinsics »?

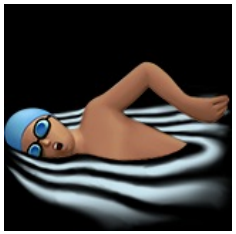
kinke commented on 1 Mar 2017 • edited ▾ Member

No idea where those 'intrinsics' came from, afaik LDC never supported these directly. Where were they declared?

Of course there's other ways to deal with SIMD; `ldc.simd` and `ldc.llvmasm` provide ways to insert textual LLVM IR and/or inline assembly (besides more generic helpers, e.g., for shuffling), giving you tremendous flexibility - via a tedious interface. ;)

Are you sure your optimizations on this low level actually pay off, i.e., does the LLVM optimizer/vectorizer not produce sufficiently efficient code with appropriate command-line options?

The builtins disappeared upstream, in clang.



Life on the other edge

Stephen Canon Jan 14, 2013; 10:37pm **Re: some sse2 intrinsics missing**

17 posts

This is a builtin, not an intrinsic. The intrinsic is _mm_cmpgt_pd.

- Steve

On Jan 14, 2013, at 4:32 PM, Richard Hadsell <[hidden email]> wrot

> It seems that Clang doesn't recognize all of the sse2 intrinsics:
>
> ```
> ./bssSIMD.h:39:9: error: use of undeclared identifier '__built:
>         r.v_ = __builtin_ia32_cmpgtpd (x, xmax.v_);
>                ^~~~~~~~~~~~~~~~~~~~~~~
> ...
> ./bssSIMD.h:51:9: error: use of undeclared identifier '__built:
>         r.v_ = __builtin_ia32_cmpltpd (x, xmin.v_);
>                ^~~~~~~~~~~~~~~~~~~~~~~
> ```

"This is a builtin, not an intrinsic"

A frequently asked question

"missing" vector __builtin functions

The Intel and AMD manuals document a number "<*mmintrin.h>" header files, which define a standardized API for accessing vector operations on X86 CPUs. These functions have names like `_mm_xor_ps` and `_mm256_addsub_pd`. Compilers have leeway to implement these functions however they want. Since Clang supports an excellent set of native vector operations, the Clang headers implement these interfaces in terms of the native vector operations.

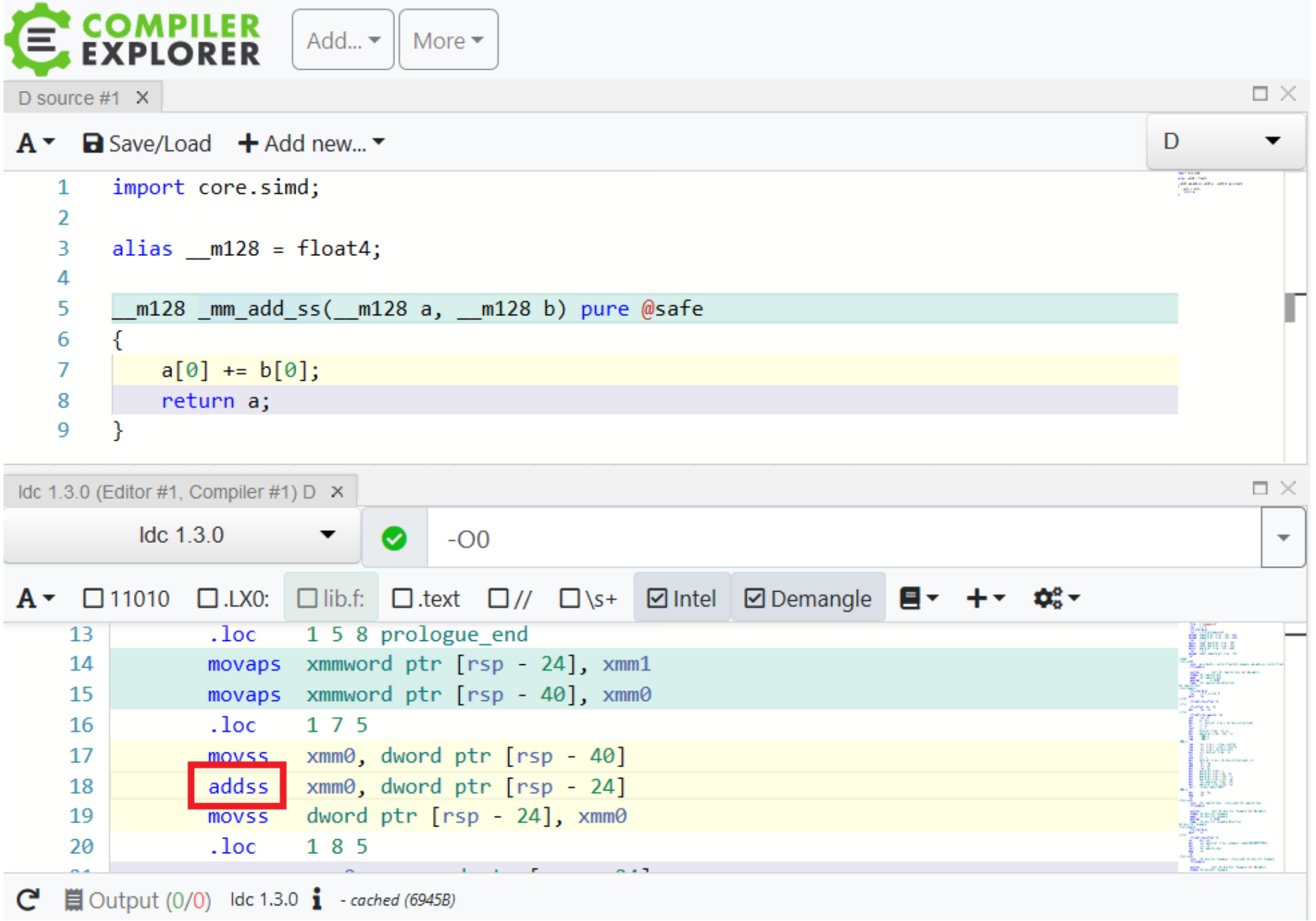
From http://clang.llvm.org/compatibility.html#vector_builtins

clang 's _mm_add_ss

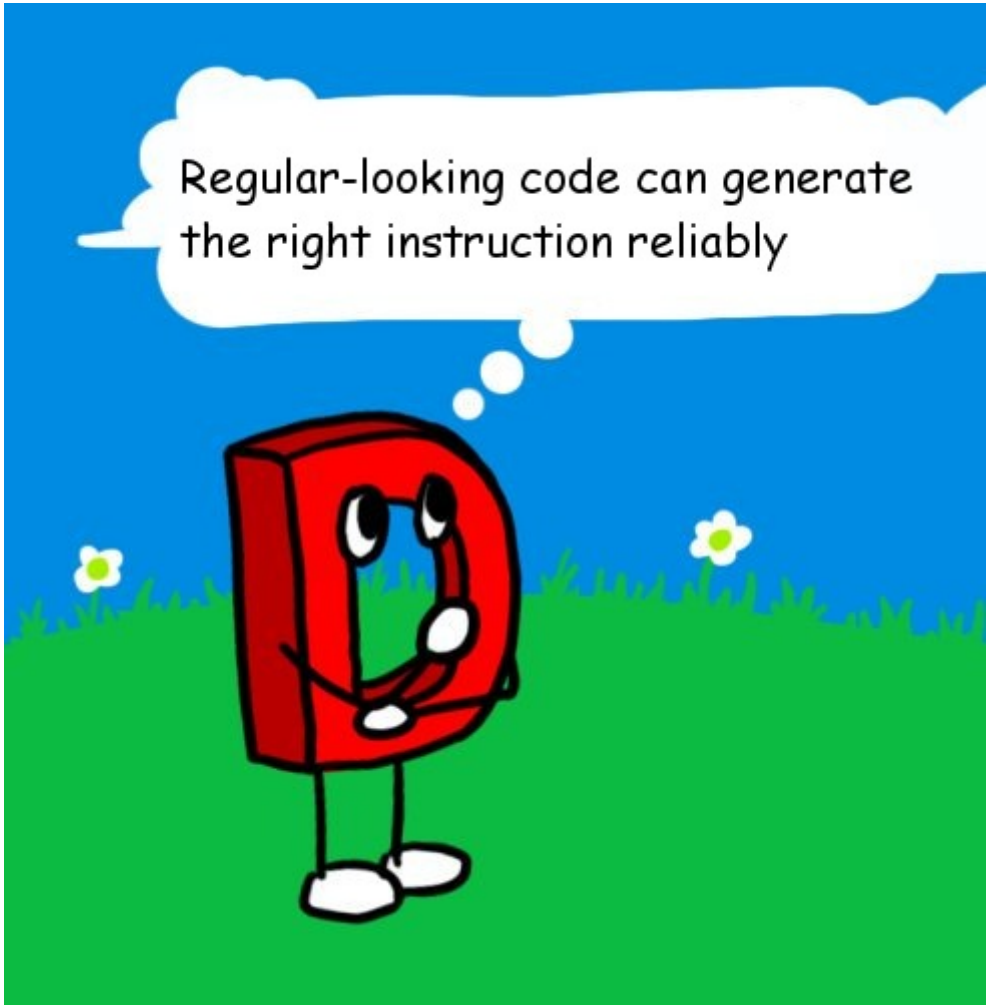
```
static __inline__ __m128 __DEFAULT_FN_ATTRS
_mm_add_ss(__m128 __a, __m128 __b)
{
    __a[0] += __b[0];
    return __a;
}
```

→ Vector extensions

Does it generate the right instruction?

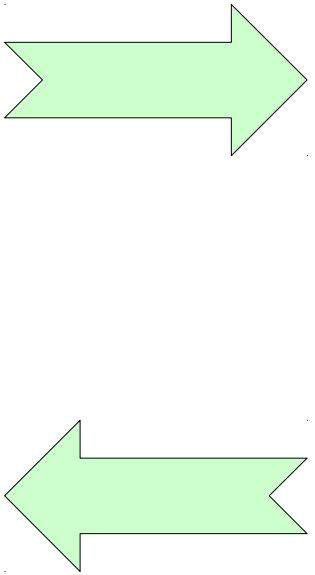


Realization #1



Realization #2

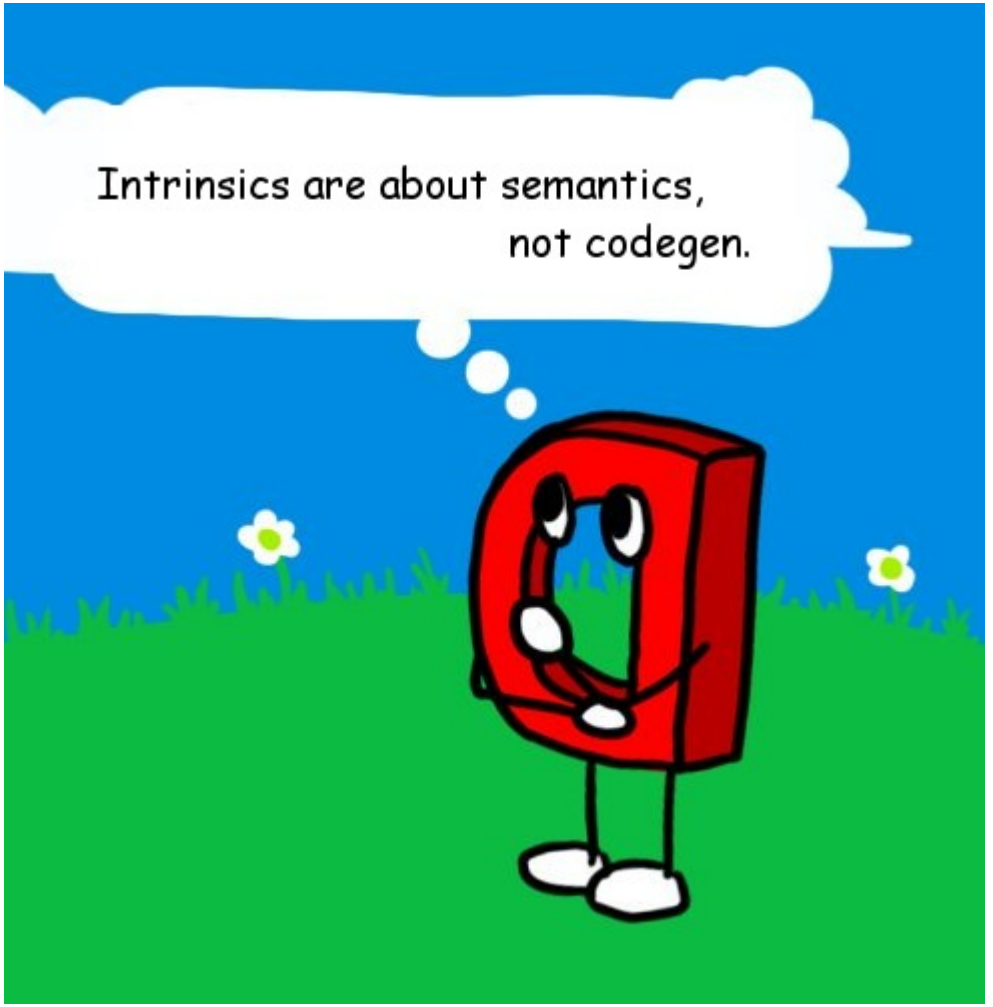
To optimize normal D code, you decide to use « intrinsics » instead of regular code to force a particular instruction



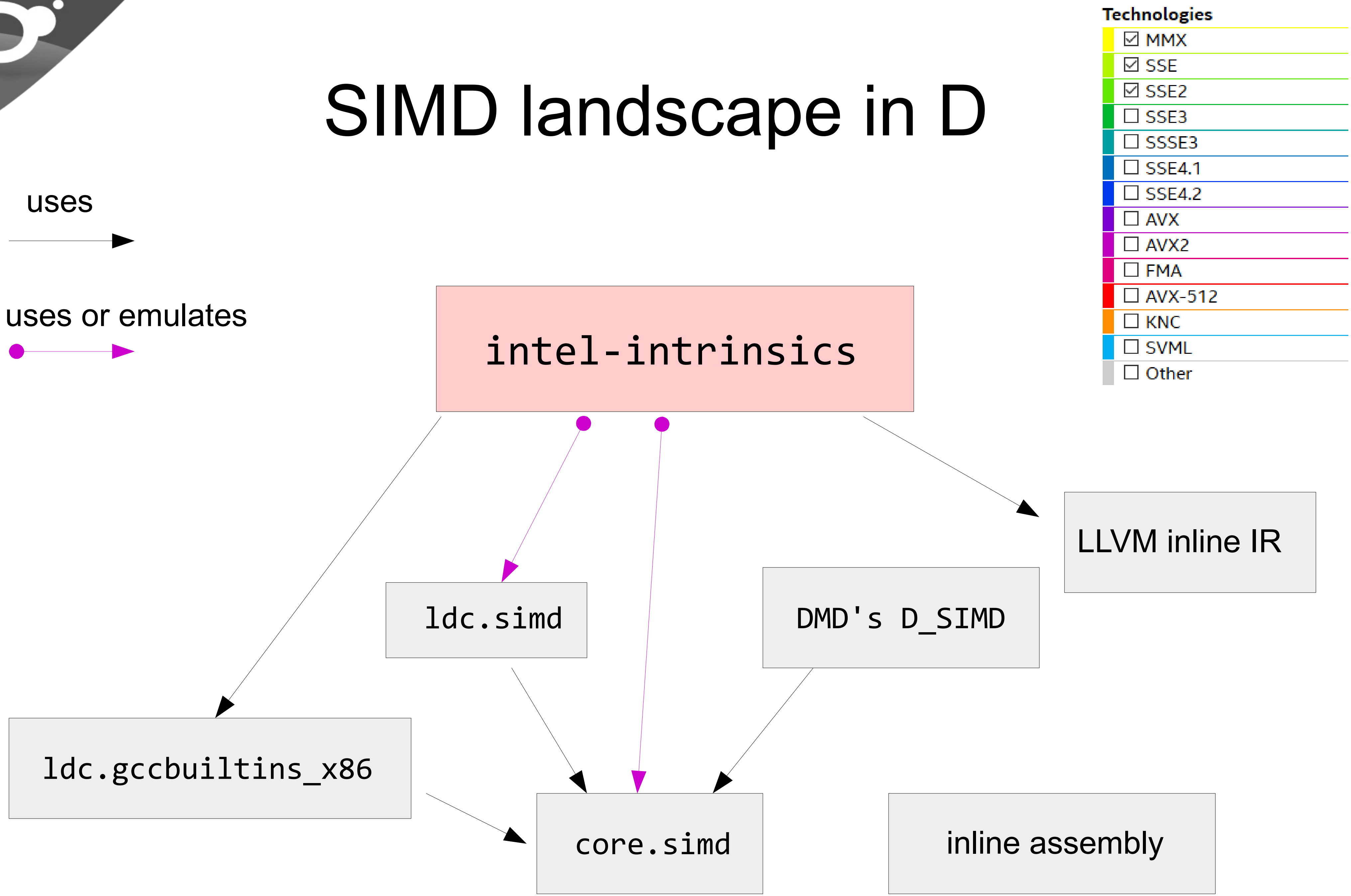
The best way to implement « intrinsics » may well be normal D code

Paradox of « intrinsics »

Realization #3



SIMD landscape in D



3 surprising things learned

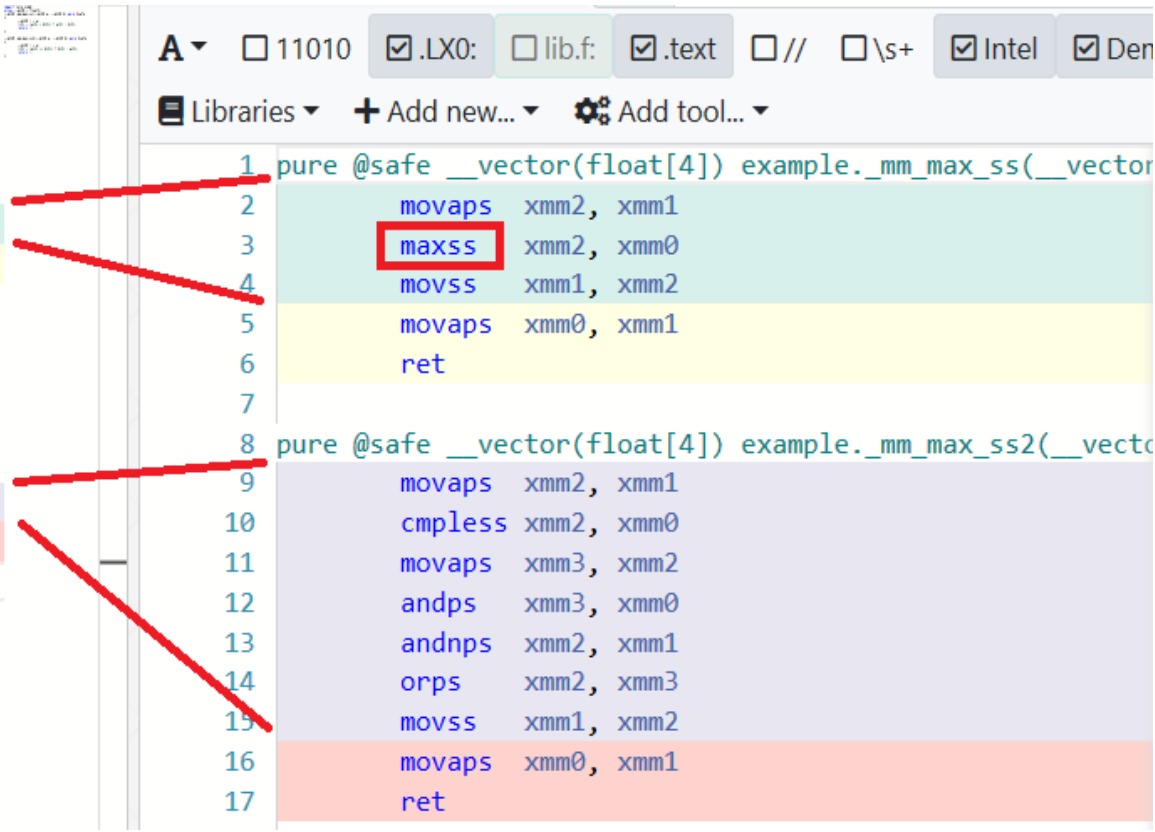
Generating PAVGW

```
__m128i _mm_avg_epu16 (__m128i a, __m128i b) pure @safe
{
    // Generates pavgw even in LDC 1.0, even in -O0
    enum ir = `
        %ia = zext <8 x i16> %0 to <8 x i32>
        %ib = zext <8 x i16> %1 to <8 x i32>
        %isum = add <8 x i32> %ia, %ib
        %isum1 = add <8 x i32> %isum, < i32 1, i32 1, i32 1, i32 1, i32 1, i32 1, i32 1, i32 1>
        %isums = lshr <8 x i32> %isum1, < i32 1, i32 1, i32 1, i32 1, i32 1, i32 1, i32 1, i32 1>
        %r = trunc <8 x i32> %isums to <8 x i16>
        ret <8 x i16> %r`;
    return cast(__m128i) LDCInlineIR!(ir, short8, short8, short8)(cast(short8)a, cast(short8)b);
}
```

Some instructions need a magic sequence of IR.

NaNs complicate everything

```d
import core.simd;
alias __m128 = float4;
__m128 _mm_max_ss(__m128 a, __m128 b) pure @safe
{
        __m128 r = a;
        r[0] = (a[0] > b[0]) ? a[0] : b[0];
        return r;
}

__m128 _mm_max_ss2(__m128 a, __m128 b) pure @safe
{
        __m128 r = a;
        r[0] = (a[0] <= b[0]) ? b[0] : a[0];
        return r;
}
```



14 ways to compare floating-point numbers, not just 4.

The deadliest cast

C++ source #1

```cpp
#include <emmintrin.h>

long long convertFloatToLongLong(__m128 a)
{
    return (long long)a[0];
}

int convertFloatToInt(__m128 a)
{
    return (int)a[0];
}
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C++

x86-64 icc 19.0.1 -O2 -m32

```
convertFloatToLongLong(__m128):
        sub       esp, 44
        movss     DWORD PTR [24+esp], xmm0
        fld       DWORD PTR [24+esp]
        fnstcw    [esp]
        movzx     eax, WORD PTR [esp]
        or        eax, 3072
        mov       DWORD PTR [8+esp], eax
        fldcw     [8+esp]
        fistp     QWORD PTR [16+esp]
        fldcw     [esp]
        mov       eax, DWORD PTR [16+esp]
        mov       edx, DWORD PTR [20+esp]
        add       esp, 44
        ret
convertFloatToInt(__m128):
        cvttss2si eax, xmm0
        ret
```

No SSE way to convert from float/double to a 64-bit integer (in 32-bit x86)

intel-intrinsics today 

- Every 516 intrinsics for SSE/SSE2/MMX
- Equivalent of `<emmintrin.h>`, `<xmmintrin.h>` and `<mmintrin.h>` but for D
- 192 `unittest`, tested on beta DMD/LDC with and without optimizations
- Some `#BONUS` intrinsics (SIMD log/exp/pow)
- Adds float2 / int2

`intel-intrinsics` today

- Same semantics for DMD and LDC (slowly emulated on DMD, mostly optimal on LDC)
- `core.simd` emulated on DMD because of Win32
- Focused on x86/x86_64 for now

`intel-intrinsics` tomorrow

- Improve performance when using DMD (leverage `core.simd` at the very least)
- Support GDC, be less LDC-exclusive
- ARM
- `pragma(inline, true)`

Disclaimer : This slide talks about future software changes

intel-intrinsics

PROS

- Brings `core.simd` when not available
- Somewhat portable, the goal is codegen decorrelated from SIMD semantics (WIP)
- Exact same results whatever the compiler
- I'm forced to maintain it

CONS

- Possibly slower debug performance
- Slower DMD performance
- Restricted to SSE/SSE2/MMX semantics

Insert that one XKCD comic about standards here

EXAMPLES

Which one is faster ?

```
dub -b release-nobounds --combined --compiler ldc2
```

```d
void squareMagnitudesNaive(const(cfloat)* complexData, float* squaredMagnitudes, int numBins)
    nothrow @nogc pure
{
    for (int bin = 0; bin < numBins; ++bin)
    {
        cfloat c = complexData[bin];
        squaredMagnitudes[bin] = c.re * c.re + c.im * c.im + 1e-10f;
    }
}

void squareMagnitudesInteli(const(cfloat)* complexData, float* squaredMagnitudes, int numBins)
    nothrow @nogc pure
{
    __m128 offset = _mm_set1_ps(1e-10f);
    for(int bin = 0; bin < numBins; bin += 2)
    {
        // read two bins at once and square them
        __m128 bins = _mm_load_ps(cast(float*)(&complexData[bin]));
        bins *= bins;
        bins += _mm_srli_ps!4(bins);
        __m128 squaredMag = _mm_shuffle_ps!0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }
}
```

Optimized code doesn't have to be ugly

```
dub -b release-nobounds --combined --compiler ldc2
```

```d
void squareMagnitudesNaive(const(cfloat)* complexData, float* squaredMagnitudes, int numBins)
    nothrow @nogc pure
{
    for (int bin = 0; bin < numBins; ++bin)
    {
        cfloat c = complexData[bin];
        squaredMagnitudes[bin] = c.re * c.re + c.im * c.im + 1e-10f;
    }
}

void squareMagnitudesInteli(const(cfloat)* complexData, float* squaredMagnitudes, int numBins)
    nothrow @nogc pure
{
    __m128 offset = _mm_set1_ps(1e-10f);
    for(int bin = 0; bin < numBins; bin += 2)
    {
        // read two bins at once and square them
        __m128 bins = _mm_load_ps(cast(float*)(&complexData[bin]));
        bins *= bins;
        bins += _mm_srli_ps!4(bins);
        __m128 squaredMag = _mm_shuffle_ps!0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }
}
```

Unrolled by 4

Unrolled by 2

2x slower

Which one is faster ?

```
dub -b release-nobounds –combined
--compiler ldc2
```

```d
import inteli.emmintrin;
import core.math;
import ldc.intrinsics: llvm_sqrt;

float distanceNaive(const(float)* a, const(float)* b) nothrow @nogc
{
    return llvm_sqrt( (a[0] - b[0])*(a[0] - b[0])
                    + (a[1] - b[1])*(a[1] - b[1])
                    + (a[2] - b[2])*(a[2] - b[2])
                    + (a[3] - b[3])*(a[3] - b[3]) );
}

float distanceInteli(const(float)* a, const(float)* b) nothrow @nogc
{
    __m128 va = _mm_loadu_ps(a);
    __m128 vb = _mm_loadu_ps(b);
    __m128 diffSquared = va - vb;
    diffSquared *= diffSquared;
    __m128 sum = _mm_add_ps(diffSquared, _mm_srli_ps!8(diffSquared));
    sum += _mm_srli_ps!4(sum);
    return _mm_cvtss_f32(_mm_sqrt_ss(sum));
}
```

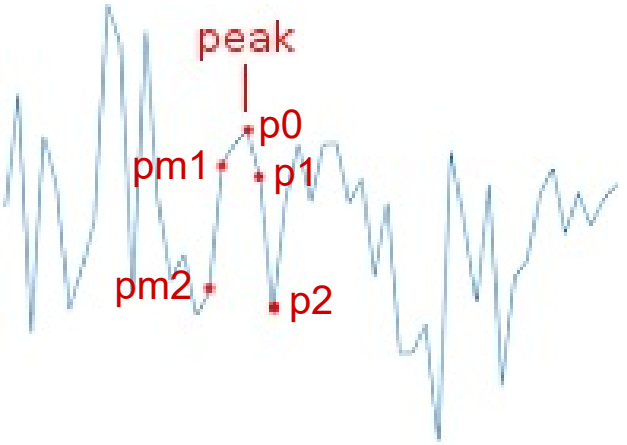
Backends are awesome

```d
import inteli.emmintrin;
import core.math;
import ldc.intrinsics: llvm_sqrt;

float distanceNaive(const(float)* a, const(float)* b) nothrow @nogc
{
    return llvm_sqrt( (a[0] - b[0])*(a[0] - b[0])
                    + (a[1] - b[1])*(a[1] - b[1])
                    + (a[2] - b[2])*(a[2] - b[2])
                    + (a[3] - b[3])*(a[3] - b[3]) );
}

float distanceInteli(const(float)* a, const(float)* b) nothrow @nogc
{
    __m128 va = _mm_loadu_ps(a);
    __m128 vb = _mm_loadu_ps(b);
    __m128 diffSquared = va - vb;
    diffSquared *= diffSquared;
    __m128 sum = _mm_add_ps(diffSquared, _mm_srli_ps!8(diffSquared));
    sum += _mm_srli_ps!4(sum);
    return _mm_cvtss_f32(_mm_sqrt_ss(sum));
}
```

equal perf

Generated code is very similar

One example that works

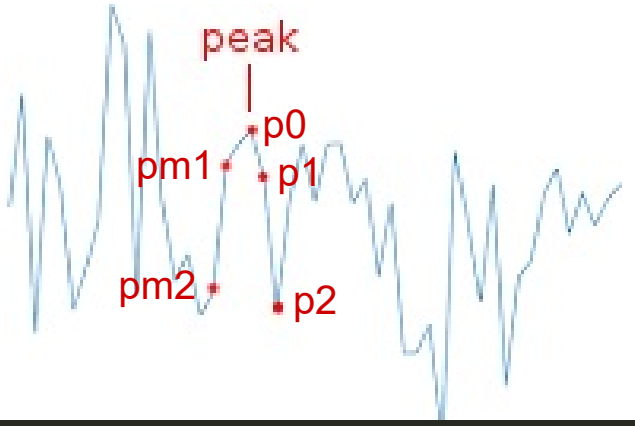



```
pm2 < pm1
pm1 < p0
p0 > p1
p1 > p2
```

```d
int countSpectralPeaksFirst(float* squaredMagnitude, int binMax)
    nothrow @nogc
{
    int numPeaks = 0;
    foreach(int bin; 2..binMax-2)
    {
        float pm2 = squaredMagnitude[bin-2];
        float pm1 = squaredMagnitude[bin-1];
        float p0  = squaredMagnitude[bin];
        float p1  = squaredMagnitude[bin+1];
        float p2  = squaredMagnitude[bin+2];

        if (pm2 < pm1 && pm1 < p0 && p0 > p1 && p1 > p2)
        {
            numPeaks += 1; // peak detected
        }
    }
    return numPeaks;
}
```

Detect spectral peaks in a phase vocoder

Using _mm_cmplt_ps and _mm_movemask_ps




```
pm2 < pm1
pm1 < p0
p0 >= p1
p1 >= p2
```

```d
int countSpectralPeaksInteli(float* squaredMagnitude, int binMax)
     nothrow @nogc
{
    import inteli.emmintrin;

    int numPeaks = 0;
    foreach(int bin; 2..binMax-2)
    {
        // pm2 pm1 p0 p1
        __m128 energy0 = _mm_loadu_ps(&squaredMagnitude[bin - 2]);
        // pm1  p0 p1 p2
        __m128 energy1 = _mm_loadu_ps(&squaredMagnitude[bin - 1]);
        // pm1<pm2 p0<pm1 p1<p0 p2<p1
        __m128 goingDown = _mm_cmplt_ps(energy1, energy0);
        int mask4bit = _mm_movemask_ps(goingDown);
        if (mask4bit == (0 + 0 + 4 + 8))
        {
            numPeaks += 1; // peak detected
        }
    }
    return numPeaks;
}
```

Benchmark results

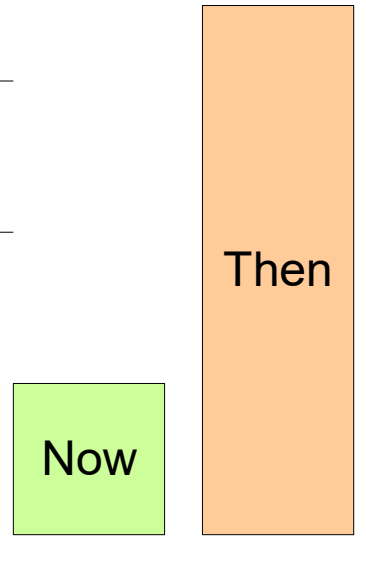
	`dub -b release-nobounds --combined`
naive	1822 ms
`intel-intrinsics`	520 ms

(ldc 1.8.0, Win64, 100000 samples)

Benchmark results

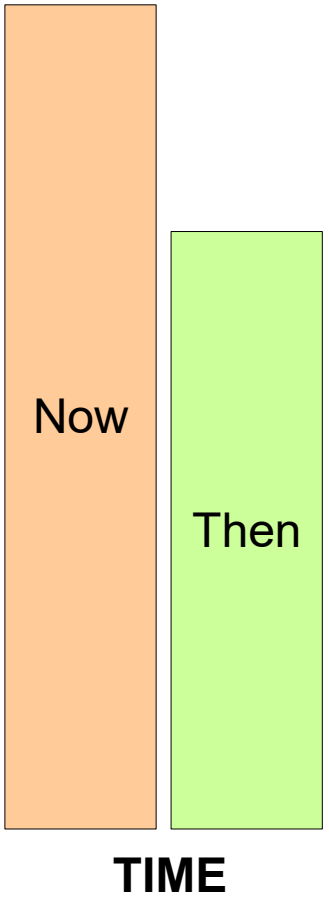
3.5x faster

	dub -b release-nobounds --combined
naive	1822 ms
intel-intrinsics	520 ms



Benchmark results

	`dub -b debug`
naive	8075 ms
`intel-intrinsics`	10981 ms

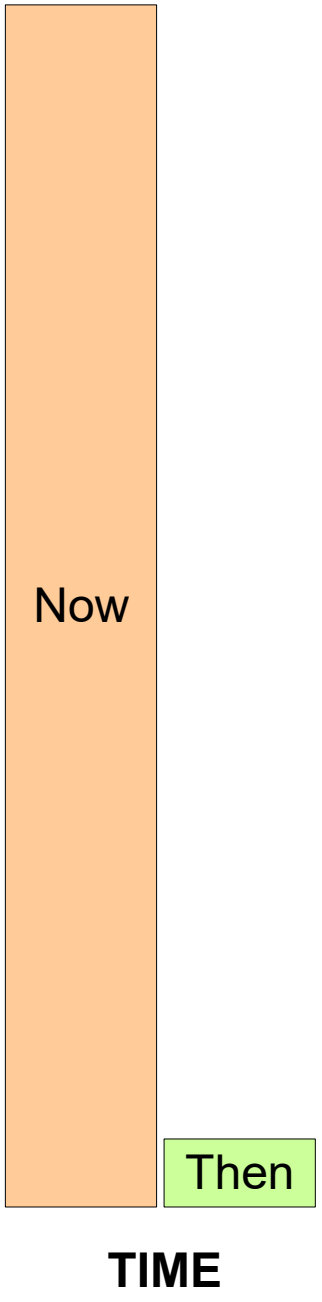


Expect worse debug performance (inlining)

(ldc 1.8.0, Win64, 100000 samples)

Benchmark results

	`dub -b release-nobounds –compiler dmd`
naive	3.307 secs
`intel-intrinsics`	65 secs



Expect worse DMD performance for now. (dmd v2.084, Win32, 100000 samples)

Take home message

A. Profile your code, measure in the following order:

- Regular D code, array ops...
- Then `intel-intrinsics`

```
"dependencies":
{
    "intel-intrinsics": "~>1.0"
}
```

B. If debug performance
OR
DMD performance is important:

➡ Maybe use both assembly and `intel-intrinsics`

C. Contributions welcome

Thank you!

Hidden content

2 ways to announce speed-ups to your boss

Hidden content

Strategy #1: Talking about Time

Baseline 600 ms	Challenger 500 ms

500 / 600 = 0.833…

1 - 500 / 600 = 0.166...

«Challenger takes **16.6 % less time** than Baseline »

Hidden content

Strategy #2: Talking about Speed

Baseline 600 ms	Challenger 500 ms

600 / 500 = 1.2

600 / 500 - 1 = 0.2

« Challenger is **20 % faster** than Baseline »

Hidden content

2 ways to announce speed-ups to your boss

« Here is a **16.6 %** improvement »

vs

« Here is a **20 %** improvement » ?

Thank you!