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

- 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

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

```c
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

```
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**

<table>
<thead>
<tr>
<th>PROS</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good x86 instruction set support</td>
<td>D_SIMD only in DMD</td>
</tr>
<tr>
<td></td>
<td>again, not in Win32</td>
</tr>
</tbody>
</table>
Option #4: ldc.simd

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

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

```d
alias __m128 = float4;

_with_core.simd:_ __m128 _mm_add_ps(__m128 a, __m128 b) pure @safe
{
    return a + b;
}
```
Implementing `_mm_add_ss`

ADDSS instruction

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

With `ldc.gccbuiltins_x86`

```cpp
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

```
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;
```
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!

These intrinsics have disappeared:

__builtin_ia32_mulss
__builtin_ia32_divss
__builtin_ia32_addss
__builtin_ia32_pmaxsw128
__builtin_ia32_pmaxub128
__builtin_ia32_pminsw128
__builtin_ia32_pminub128'
__builtin_ia32_pshufD
__builtin_ia32_pshufhw
__builtin_ia32_pshufld
__builtin_ia32_storel4v4si
__builtin_ia32_storedqu
__builtin_ia32_storupd

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 »?

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 »?

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

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]> wrote

It seems that Clang doesn't recognize all of the sse2 intrinsics:

```c
./bssSIMD.h:39:9: error: use of undeclared identifier '___built:
   r.v_ = ___builtin_i32_cmpgtpd (x, xmax.v_);
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```c
./bssSIMD.h:51:9: error: use of undeclared identifier '___built:
   r.v_ = ___builtin_i32_cmpltpd (x, xmin.v_);
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

"This is a builtin, not an intrinsic" 🤷‍♂️
"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

```c
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

Regular-looking code can generate the right instruction reliably.
Realization #2

Builtins are a last resort strategy.
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

Intrinsics are about semantics, not codegen.
SIMD landscape in D

- **intel-intrinsics**
  - **ldc.simd**
  - **DMD's D_SIMD**
  - **LLVM inline IR**
  - **forall**
  - **ldc.gccbuiltins_x86**
  - **core.simd**
  - **inline assembly**

**Technologies**
- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

**Uses**
- **uses**

**Uses or Emulates**
- **uses or emulates**
3 surprising things learned
Generating PAVGw

```c
__m128i _mm_avg_epu16 (__m128i a, __m128i b) pure @safe
{
    // Generates pavgw even in LDC 1.0, even in -00
    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

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

No SSE way to convert from float/double to a 64-bit integer (in 32-bit x86)
- Every 516 intrinsics for SSE/SSE2/MMX
- Equivalent of `<emmintrin.h>`, `<xmmmintrin.h>` and `<mmintrin.h>` but for D
- 192 unit test, tested on beta DMD/LDC with and without optimizations
- Some `#BONUS` intrinsics (SIMD log/exp/pow)
- Adds float2 / int2
intell-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) noexcept @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) noexcept @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_srl_epu4(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

void squareMagnitudesNaive(const(cfloat)* complexData, float* squaredMagnitudes, int numBins)
    nothrow @nogc pure
{
    for (int bin = 0; bin < numBins; ++bin) // Unrolled by 4
    {
        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); // Unrolled by 2
    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_srl_ps4(bins);
        __m128 squaredMag = _mm_shuffle_ps0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }
}
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])
}

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_slli_ps!8(diffSquared));
    sum += _mm_slli_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])
}

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_srlkap18(diffSquared));
    sum += _mm_srlki_ps14(sum);
    return _mm_cvtss_f32(_mm_sqrt_ss(sum));
}
```

Generated code is very similar
One example that works

```c
int countSpectralPeaksFirst(float* squaredMagnitude, int binMax) {
    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_cmpllt_ps and _mm_movemask_ps

```c
int countSpectralPeaksInteli(float* squaredMagnitude, int binMax)
{
    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_cmpllt_ps(energy1, energy0);
        int mask4bit = _mm_movemask_ps(goingDown);
        if (mask4bit == (0 + 0 + 4 + 8))
        {
            numPeaks += 1; // peak detected
        }
    }
    return numPeaks;
}
```
### Benchmark Results

<table>
<thead>
<tr>
<th></th>
<th><code>dub -b release-nobounds --combined</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>1822 ms</td>
</tr>
<tr>
<td>intel-intrinsics</td>
<td>520 ms</td>
</tr>
</tbody>
</table>

(ldc 1.8.0, Win64, 100000 samples)
Benchmark results

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>1822 ms</td>
</tr>
<tr>
<td>intel-intrinsics</td>
<td>520 ms</td>
</tr>
</tbody>
</table>

3.5x faster

Now

Then
Expect worse debug performance (inlineing) (lde 1.8.0, Win64, 100000 samples)
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

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
Strategy #1: Talking about Time

Baseline: 600 ms
Challenger: 500 ms

\[
\frac{500}{600} = 0.833\ldots
\]

\[
1 - \frac{500}{600} = 0.166\ldots
\]

«Challenger takes 16.6 % less time than Baseline »
Hidden content

Strategy #2: Talking about Speed

Baseline: 600 ms
Challenger: 500 ms

$\frac{600}{500} = 1.2$

$\frac{600}{500} - 1 = 0.2$

« Challenger is **20 % faster** than Baseline »
2 ways to announce speed-ups to your boss

« Here is a 16.6 % improvement »

vs

« Here is a 20 % improvement » ?
Thank you!