Practical SIMD acceleration with Boost.SIMD

Mathias Gaunard Joël Falcou Jean-Thierry Lapresté

MetaScale

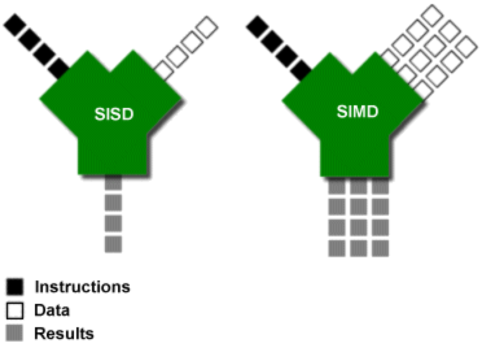
Boostcon 2011

Context

From NT^2 to Boost.SIMD

- Last year, we presented NT^2, a MATLAB-like Proto-based library for high-performance numerical computation
- Boost.SIMD is the extraction of the SIMD subcomponent of the library
- A GSoC project is scheduled this summer to help make it ready for review
- This talk is here to present what's inside the proposal.

What's SIMD?



Principles

- Single Instruction, Multiple Data
- Operations applied on NxT elements within a single register
- Up to N times faster than regular ALU/FPU

Why is SIMD abstraction needed?

x86 family

- MMX 64-bit float, double
- SSE 128-bit float
- SSE2 128-bit int8, int16, int32, int64, double
- SSE3
- SSSE3
- SSE4a (AMD only)
- SSE4.1
- SSE4.2
- AVX 256-bit float, double
- FMA4 (AMD only)
- XOP (AMD only)
- FMA3

PowerPC family

- AltiVec 128-bit int8, int16, int32, int64, float
- Cell SPU 128-bit int8, int16, int32, int64, float, double

ARM family

- VFP 64-bit float, double
- NEON 64-bit and 128-bit float, int8, int16, int32, int64

Why not let the compiler do it?

Compilers are only so smart

- Automatic vectorization can only happen if:
  - Memory is well agenced
  - Code is inherently vectorizable
- Compiled functions are not vectorized (I look at you libm !)
- Compilers don't always have enough static information to know what they can vectorize
- Designing for vectorization is a human process

Conclusion

- Declaring SIMD parallelism explicitly is the best way to get your code vectorized
- To be demonstrated by this presentation

Talk Layout

Writing it by hand

Doing a * b + c with vectors of 32-bit integers : SSE

```
__m128i a, b, c, result;
result = _mm_mul_epi32(a, _mm_add_epi32(b, c));
```

Doing a * b + c with vectors of 32-bit integers : Altivec

```
__vector int a, b, c, result;
result = vec_cts(vec_madd( vec_ctf(a,0)
                         , vec_ctf(b,0)
                         , vec_ctf(c,0)
                         )
                ,0);
```

The pack abstraction

simd::pack<T>

pack<T, N>	SIMD register that packs N elements of type T
pack<T>	automatically finds best N available

- Behaves just like T except operations yield a pack of T and not a T.

Constraints

- T must be a fundamental arithmetic type, i.e. (un)signed char, (unsigned) short, (unsigned) int, (unsigned) long, (unsigned) long long, float or double – not bool.
- N must be a power of 2.

pack API

Operators

- All overloadable operators are available
- `pack<T>` x `pack<T>` operations but also `pack<T>` x `T`
- Type coercion and promotion disabled
  `uint8_t(255)+ uint8_t(1)` yields `uint8_t(0)`, not `int(256)`

Comparisons

- `==`, `!=`, `<`, `<=`,`>` and `>=` perform lexical comparisons.
- `eq`,`neq`,`lt`,`gt`,`le` and `ge` as functions return `pack` of boolean.

Other properties

- Models both a `ReadOnlyRandomAccessFusionSequence` and `ReadOnlyRandomAccessRange`
- `at_c<i>(p)` or `p[i]` can be used to access the i-th element, but is usually slow (`at_c` is faster)

pack API

Memory access

Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from or to a `T*`. Errors leads to undefined behaviors.

Examples

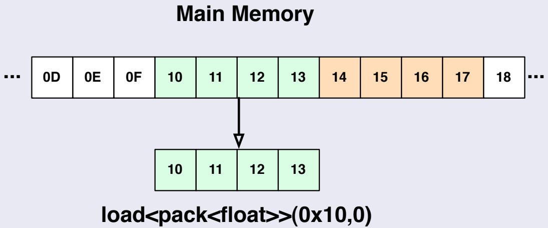
pack API

Memory access

Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from or to a `T*`. Errors leads to undefined behaviors.

Examples

`load< pack<T, N> >(p, i)` loads pack at aligned address `p + i*N`



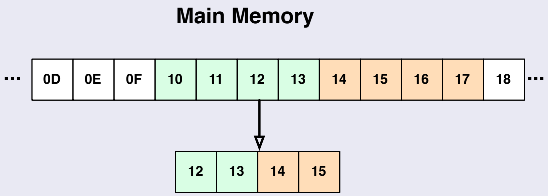
pack API

Memory access

Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from or to a `T*`. Errors leads to undefined behaviors.

Examples

`load< pack<T, N>, Offset>(p, i)` loads pack at address `p + i*N + Offset`, `p + i` must be aligned.



load<pack<float>,2>(0x10,0)

pack API

Memory access

Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from or to a `T*`. Errors leads to undefined behaviors.

Examples

`store(p, i, pk)` stores pack `pk` at aligned address `p + i*N`

pack as a proto entity

Rationale

- Most SIMD ISA have fused operations (FMA, etc...)
- We want to write simple code but yet get best performances out of these
- We need lazy evaluation : `proto` to the rescue !

Advantage

- All expressions, even those involving functions, generate template expressions that are evaluated on assignment or in the conversion operator
- `a * b + c` is mapped to `fma(a, b, c)`
  `a + b * c` is mapped to `fma(b, c, a)`
  `!(a < b)` is mapped to `is_nle(a, b)`
- the optimisation system is open for extensions

Extra arithmetic, bitwise and ieee operations, predicates

Arithmetic

- saturated arithmetic
- float/int conversion
- round, floor, ceil, trunc
- sqrt, hypot
- average
- random
- min/max
- rounded division and remainder

Bitwise

- select
- andnot, ornot
- popcnt
- ffs
- ror, rol
- rshr, rshl
- twopower

IEEE

- ilogb, frexp
- ldexp
- next/prev
- ulpdist

Predicates

- comparison with zero
- negation of comparison
- is_unord, is_nan, is_invalid
- is_odd, is_even
- majority

Reduction and SWAR operations

Reduction

- any, all
- nbtrue
- minimum/maximum, posmin/posmax
- sum
- product, dot product

SWAR

- group/split
- splatted reduction
- cumsum
- sort

`native<T, X>` : SIMD register of `T` on arch. `X`

Semantic

- like `pack` but Plain Old Data and all operations and functions return values and not expression templates.
- `X` characterizes the register type, not the instructions available. Only one tag for all SSE variants.
- It is the interface that must be used to extend the library.

Example

`native<float, tag::sse_>`	wraps a `__m128`
`native<uint8_t, tag::sse_>`	wraps a `__m128i`
`native<double, tag::avx_>`	wraps a `__m256d`
`native<float, tag::altivec_>`	wraps a `__vector float`

`native<T, X>` : SIMD register of `T` on arch. `X`

Software fallback

- `tag::none_<N>` is a software-emulated SIMD architecture with a register size of `N` bytes
- It is used as fallback when no satisfying SIMD architecture is found
- Thanks to this, code can degrade well and remain portable.
- Default native type when no SIMD is found : `native<T, tag::none_<8> >`

RGB to grayscale

Scalar version

```
float const *red, *green, *blue;
float* result;

for(std::size_t i = 0; i != height*width; ++i)
    result[i] = 0.3f * red[i] + 0.59f * green[i] + 0.11f * blue[i];
```

SIMD version

```
std::size_t N = meta::cardinal_of<pack<float>>::value;
for(std::size_t i = 0; i != height*width/N; ++i)
{
   pack<float> r = load< pack<float> >(red, i);
   pack<float> g = load< pack<float> >(green, i);
   pack<float> b = load< pack<float> >(blue, i);

   pack<float> res = 0.3f * r + 0.59f * g + 0.11f * b;
   store(res, result, i);
}
```

Easy enough, but what if...

- ... I've got interleaved RGB or RGBA?
- ... I've got 8-bit integers and not floats?

Sounds more complicated, we'll see that later.

Talk Layout

Operations vs Data

Where/How to store our data ?

- SIMD operations require data to operate onto
- Usual approaches force a specific container type onto users
- Not generic enough

A better approach

- SIMD compliant allocators
- SIMD Range and Iterators over ContiguousRange
- Adapt our SIMD classes to work with a subset of STD algorithms

Operations vs Data

Where/How to store our data ?

- SIMD operations require data to operate onto
- Usual approaches force a specific container type onto users
- Not generic enough

A better approach

- SIMD compliant allocators
- SIMD Range and Iterators over ContiguousRange
- Adapt our SIMD classes to work with a subset of STD algorithms

SIMD allocators

Rationale

- Allow containers to handle memory in a SIMD compliant way
- Handles alignment of memory
- Handles padding of memory

Example

```
std::vector<float, simd::allocator<float> > v(173);

assert( simd::is_aligned(&v[0]) );
```

From Range to SIMDRange

Iterator interface

- Boost.SIMD provides `simd::begin()`/`simd::end()`
- Turn iterators into SIMD iterators returning `pack`
- Take a regular range, iterate over it in SIMD

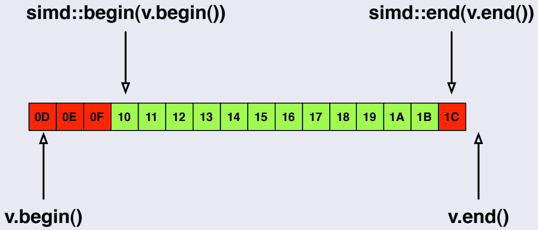
Example

From Range to SIMDRange

Iterator interface

- Boost.SIMD provides `simd::begin()`/`simd::end()`
- Turn iterators into SIMD iterators returning `pack`
- Take a regular range, iterate over it in SIMD

Example



From Range to SIMDRange

Iterator interface

- Boost.SIMD provides `simd::begin()`/`simd::end()`
- Turn iterators into SIMD iterators returning `pack`
- Take a regular range, iterate over it in SIMD

Example

```
std::vector<float, simd::allocator<float> > v(1024);
pack<float> x,z;

x = std::accumulate( simd::begin(v.begin())
                   , simd::end(v.end())
                   , z
                   );
```

From Range to SIMDRange

Iterator interface

- Boost.SIMD provides `simd::begin()`/`simd::end()`
- Turn iterators into SIMD iterators returning `pack`
- Take a regular range, iterate over it in SIMD

Example

```
std::vector<float, simd::allocator<float> > v(1024);
pack<float> x,z;

x = boost::accumulate(simd::range(v), z);
```

From Range to SIMDRange

Iterator interface

- `native` and `pack` provides `begin()/end()`
- Directly usable in STD algorithms
- Directly usable in Boost.Range algorithms

Example

```
pack<float> x(1,2,3,4);

float k = std::accumulate(x.begin(), x.end(), 0.f);
```

SIMD values as Range

Putting everything together

```
std::vector<float, simd::allocator<float> > v(1024);
pack<float> x,z;
float r;

x = boost::accumulate(simd::range(v), z);
r = std::accumulate(x.begin(), x.end(), 0.f);
```

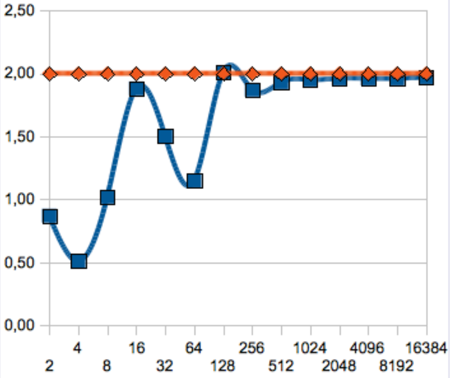
SIMD values as Range

Putting everything together - Better version

```
std::vector<float, simd::allocator<float> > v(1024);
float r;

r = sum(accumulate(simd::range(v),pack<float>()));
```

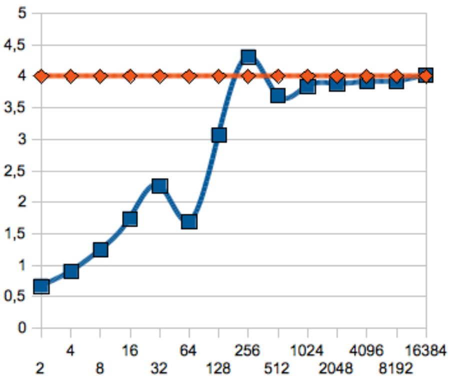
SIMD values as Range

std::accumulate speed-up for double



SIMD values as Range

`std::accumulate` speed-up for `float`



SIMD Range and generic SIMD/scalar code

Back to RGB2Grey

```
template<class RangeIn, class RangeOut> inline void
rgb2grey( RangeOut result, RangeIn red, RangeIn green, RangeIn blue )
{
  typedef typename RangeIn::iterator  in_iterator;
  typedef typename RangeOut::iterator iterator;
  typedef typename iterator_value<iterator>::type type;

  iterator br = result.begin(), er = result.end();
  in_iterator r = red.begin();
  in_iterator g = green.begin();
  in_iterator b = blue.begin();

  while( br != er )
  {
    type rv = load< type >(r, 0);
    type gv = load< type >(g, 0);
    type bv = load< type >(b, 0);
    type res = 0.3f * rv + 0.59f * gv + 0.11f * bv;
    store(res, br, 0);
    br++; r++; g++; b++;
  }
}
```

What's Missing ?

Integrated SIMD support

- Most STD algorithms should be specialized to be run in one scoop
- Can we have a Boost.Range adaptor like `simd(r)` ?
- Support for shifted Range using `load<T,N>`

Some SIMD mind teasers

- SIMD `find` ?
- SIMD `sort` ?
- Accelerating stuff like `copy` ?

Talk Layout

Boolean values in SIMD

The Problem

```
pack<float> x(1,2,3,4);
pack<float> c(2.5);

cout << lt(x,c) << endl;
```

Boolean values in SIMD

The Problem

```
pack<float> x(1,2,3,4);
pack<float> c(2.5);

cout << lt(x,c) << endl;
(( Nan Nan 0  0))
```

Boolean values in SIMD

The Problem

```
pack<float> x(1,2,3,4);
pack<float> c(2.5);

cout << lt(x,c) << endl;
(( Nan Nan 0  0))
```

The Solution

```
True<T>() which returns a proper true value w/r to T
False<T>() which returns a proper false value w/r to T
```

Conditionnal in SIMD

Example

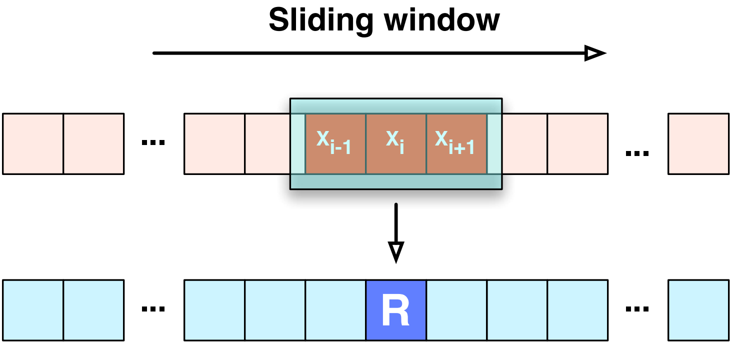
```
// Scalar code
if( x > 4 )
  y = 2*x;
else
  z = 1.f/x

// SIMD code
// ???
```

Conditionnal in SIMD

Example

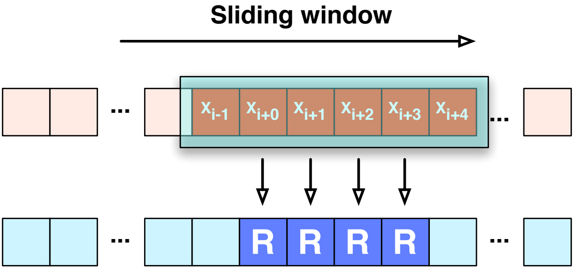
```
// Scalar code
if( x > 4 )
  y = 2*x;
else
  z = 1.f/x

// SIMD code
y = where( gt(x, 4), 2*x, y);
z = where( gt(x, 4), z, 1.f/x);
```

Motivation



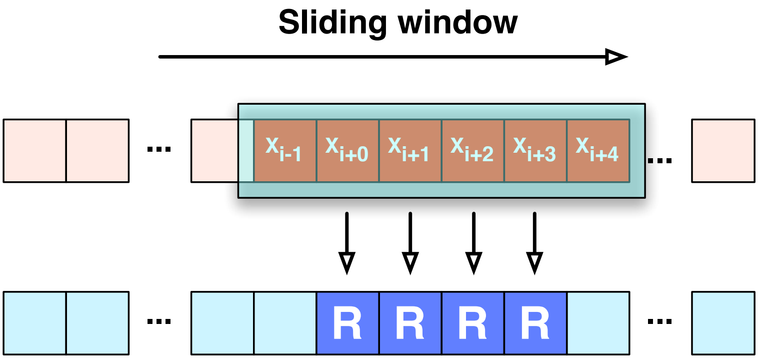
$$R[i] = 1/3 * (x[i-1] + x[i] + x[i+1])$$

Getting there ...



$$R[4*i+0] = 1/3 * (x[4*i-1] + x[4*i+0] + x[4*i+1])$$
$$R[4*i+1] = 1/3 * (x[4*i+0] + x[4*i+1] + x[4*i+2])$$
$$R[4*i+2] = 1/3 * (x[4*i+1] + x[4*i+2] + x[4*i+3])$$
$$R[4*i+3] = 1/3 * (x[4*i+2] + x[4*i+3] + x[4*i+4])$$

The vector solution




```
VR = 1/3 * ( load<-1>(vx) + load<0>(vx) + load<1>(vx) )
```

The vector solution

SIMD/Scalar version

```
template<class RangeIn, class RangeOut>
inline void average( RangeOut result, RangeIn input )
{
  typedef typename RangeIn::iterator  in_iterator;
  typedef typename RangeOut::iterator iterator;
  typedef typename iterator_value<iterator>::type type;

  iterator br = result.begin(), er = result.end();
  in_iterator data = input.begin();

  br++; er--;
  while( br != er )
  {
    type xm1 = load<type, -1>(data,i);
    type x      = load<type>(data,i);
    type xp1  = load<type, +1>(data,i);

    store(res,i,0) = 1.f/3 * (xm1 + x + xp1);
  }
}
```

Back to RGB2Grey

8-bit RGB

```
static const std::size_t N = meta::cardinal_of< pack<uint8_t> >::value;
for(std::size_t i = 0; i != height*width/N; ++i)
{
  pack<uint8_t> r = load< pack<uint8_t> >(red, i);
  pack<uint8_t> g = load< pack<uint8_t> >(green, i);
  pack<uint8_t> b = load< pack<uint8_t> >(blue, i);

  pack<uint8_t> res = uint8_t(77) * r / uint8_t(255) + uint8_t(150)
                    * g / uint8_t(255) + uint8_t(28) * b / uint8_t(255);
  store(res, result, i);
}
```

Back to RGB2Grey

Promote the pack

```
uint16_t r_coeff = 77;
uint16_t g_coeff = 150;
uint16_t b_coeff = 28;
uint16_t div_coeff = 255;

pack<uint16_t> r1, r2, g1, g2, b1, b2;
tie(r1, r2) = split(r);
tie(g1, g2) = split(g);
tie(b1, b2) = split(b);

pack<uint16_t> res1 = (r_coeff * r1+ g_coeff * g1 + b_coeff * b1) / div_coeff;
pack<uint16_t> res2 = (r_coeff * r2+ g_coeff * g2 + b_coeff * b2) / div_coeff;
pack<uint8_t> res = group(res1, res2);
```

Talk Layout

metascale

Overview of Boost.SIMD

Our goals

- Bring SIMD programing to a usable state
- if we have `boost.atomic`, why not `boost.simd` ?
- Be attractive by being nice with the rest of C++

What we achieved

- Leveraging what we learned in NT^2
- Demonstrated some impacts in term of performance
- Made using SIMD almost as simple than scalar

Upcoming works

Google Summer of Code 2011

- Cleanign up the mess and boostify it
- Improve STL/Boost compatibility
- Wanted: Applications so we can have real life examples in the library

Thanks for your attention