The Numerical Template Toolbox or *Why We Stopped Worrying & Loved Boost::Proto*

Joel Falcou and Jean-Thierry Lapresté

LRI, University Paris Sud XI
LASMEA, University Blaise Pascal - Clermont-Ferrand II

BOOST'CON 2010

Context

In Scientific Computing ...

- there is **Scientific**

Context

In Scientific Computing ...

- there is **Scientific**
  - Applications are domain driven
  - Users $\neq$ Developers
  - Users are reluctant to changes

Context

In Scientific Computing ...

- there is **Scientific**
  - Applications are domain driven
  - Users $\neq$ Developers
  - Users are reluctant to changes
- there is **Computing**

Context

In Scientific Computing ...

- there is **Scientific**
  - Applications are domain driven
  - Users $\neq$ Developers
  - Users are reluctant to changes
- there is **Computing**
  - Computing requires performance ...
  - ... which implies architectures specific tuning
  - ... which requires expertise
  - ... which may or may not be available

Context

In Scientific Computing ...

- there is **Scientific**
  - Applications are domain driven
  - Users $\neq$ Developers
  - Users are reluctant to changes
- there is **Computing**
  - Computing requires performance ...
  - ... which implies architectures specific tuning
  - ... which requires expertise
  - ... which may or may not be available

The Problem

People **using** computers to do science want to do **science** first.

The Users Conundrum

Alice, the Incidental User

- is a scientist, but not a computer scientist
- use some computing tools (MATLAB™, Maple, R ...)

Bob, the Average Developer

- has expertise in a couple of high-level languages (C++, JAVA)
- don't want to waste time on **just** the computing part of his work

Carole, the Hardcore Expert

- knows her machines down to the wires
- wants a fine grained control over her code

The Problem – and how we want to solve it

The Facts

- The **"Library to bind them all"** doesn't exist (*or we should have it already*)
- All those users want to take advantage of new architectures
- Few of them want to actually handle all the dirty work

The Ends

- Provide a "familiar" interface that let users benefit from parallelism
- Let power-users be able to fine tune whatever they fancy
- Helps compilers to generate better parallel code

The Means

- Generic Programming
- Template Meta-Programming
- Embedded Domain Specific Languages

Talk Layout

What is NT^2 ?

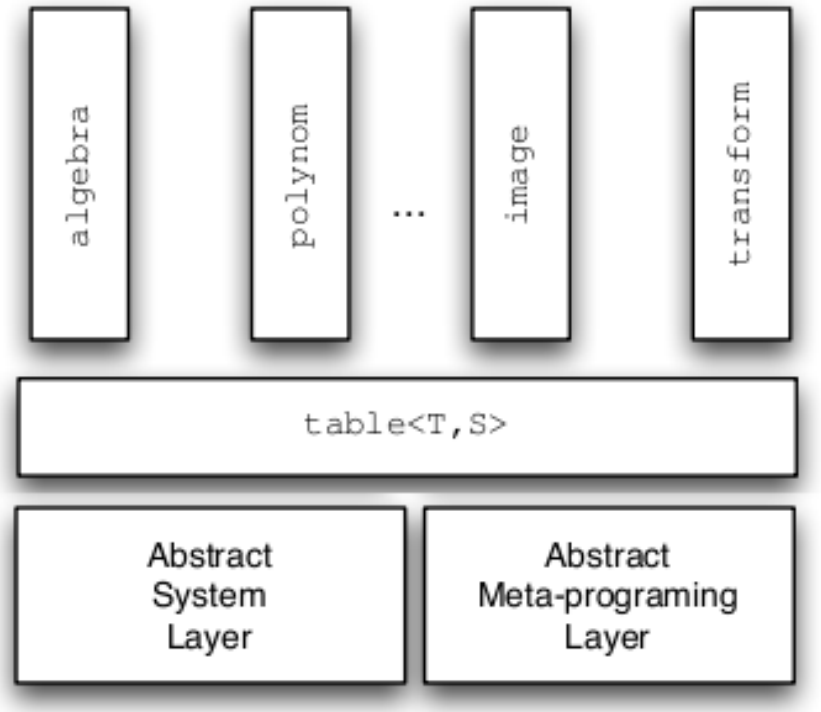
A Scientific Computing Library

- Provide a simple, MATLAB-like interface for users
- Provide high-performance computing entities and primitives
- Easily extendable

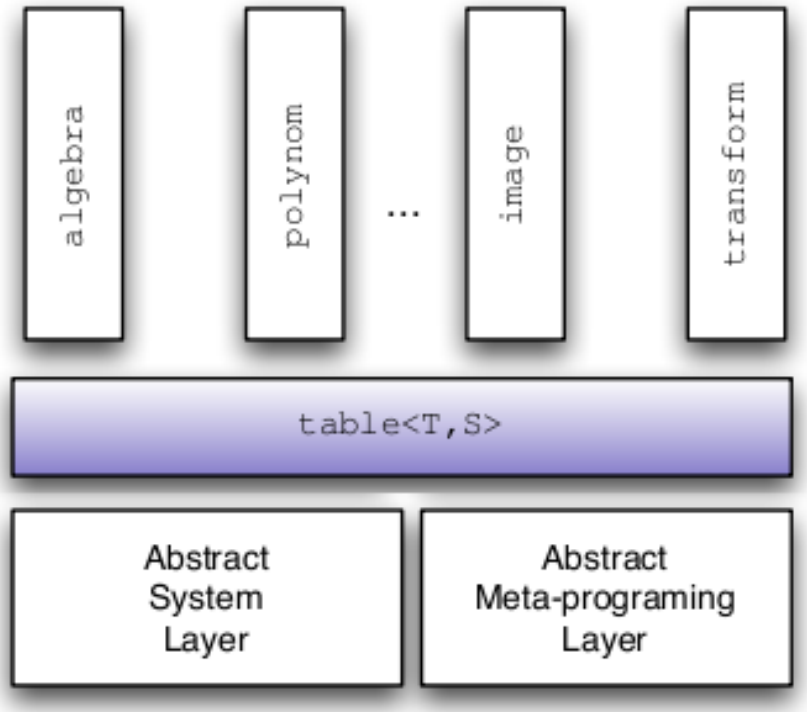
A Research Platform

- Simple framework to add new optimization schemes
- Test bench for EDSL development methodologies
- Test bench for Generic Programming in real life projects

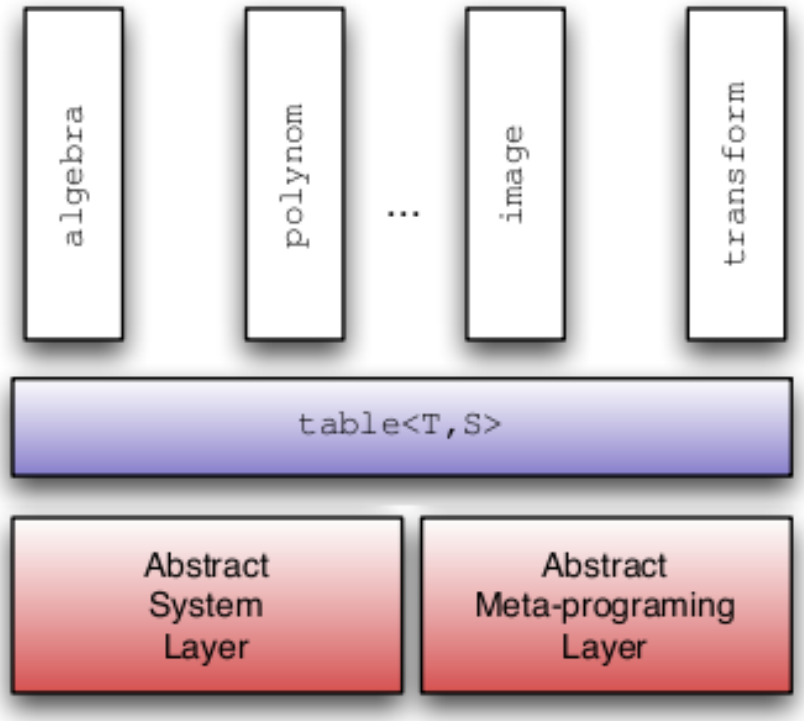
NT^2 Structure



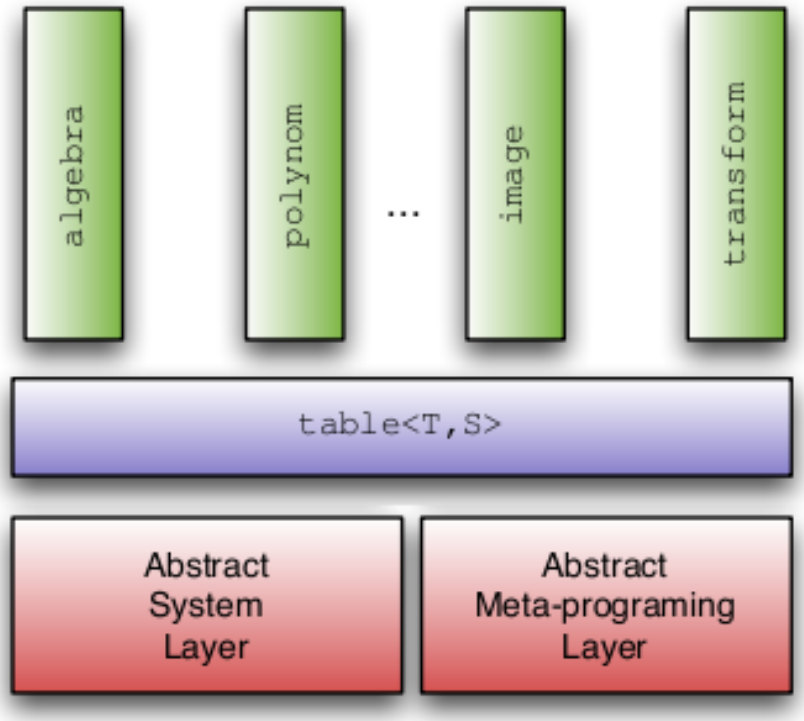
NT^2 Structure



NT^2 Structure



NT^2 Structure



The NT^2 API

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics MATLAB array behavior and functionalities
- Most mathematical functions are usable directly either on `table` or on any scalar values as in MATLAB

The NT^2 API

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics MATLAB array behavior and functionalities
- Most mathematical functions are usable directly either on `table` or on any scalar values as in MATLAB

How does it works

- Take a `.m` file, copy to a `.cpp` file

The NT^2 API

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics MATLAB array behavior and functionalities
- Most mathematical functions are usable directly either on `table` or on any scalar values as in MATLAB

How does it works

- Take a `.m` file, copy to a `.cpp` file
- Add `#include <nt2/nt2.hpp>` and do cosmetic changes

The NT^2 API

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics MATLAB array behavior and functionalities
- Most mathematical functions are usable directly either on `table` or on any scalar values as in MATLAB

How does it works

- Take a `.m` file, copy to a `.cpp` file
- Add `#include <nt2/nt2.hpp>` and do cosmetic changes
- Compile the file and link with `libnt2.a`

The NT^2 API

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics MATLAB array behavior and functionalities
- Most mathematical functions are usable directly either on `table` or on any scalar values as in MATLAB

How does it works

- Take a `.m` file, copy to a `.cpp` file
- Add `#include <nt2/nt2.hpp>` and do cosmetic changes
- Compile the file and link with `libnt2.a`
- ???

The NT^2 API

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics MATLAB array behavior and functionalities
- Most mathematical functions are usable directly either on `table` or on any scalar values as in MATLAB

How does it works

- Take a `.m` file, copy to a `.cpp` file
- Add `#include <nt2/nt2.hpp>` and do cosmetic changes
- Compile the file and link with `libnt2.a`
- ???
- **PROFIT !**

The NT^2 API

Separation between MATLAB and C++

- MATLAB-like operations are free functions
- Natural C++ operations are (mostly) member functions
- Each of them works as expected in their relative context

Example

```
table<double> m( of_size(5,7,3) );

// Returns 105
table<double>::size_type s = m.size();

// Returns (5,7,3)
table<double>::dimension_type z = size(m);
```

The NT^2 API

Specific C++ support

- Containers model the adequate concepts
- Functions support iterators and ranges
- **NT^2** is interoperable with the STL and `Boost` components

Specific MATLAB support

- Index starts at 1 (unless specified otherwise)
- Tables are stored in column major mode (FORTRAN compatibility)
- *n–dims* table can be intuitively manipulated as a *m–dims* table
- _ represents a whole range (in place of :)

The NT^2 API

Matrix Settings

- No way to get the perfect numeric container with all features
- **Compile-Time Policies** handle variability
- Traits classes ease user definition of selection among containers

Supported settings

- `shallow_,shared_`: table acts as views
- `of_size_<...Dims>`: compile-time sized container
- `storage_<...Dims>`: storage-mode ordering
- `index_<...Dims>`: base index per dimensions
- `id_<'name'>`: compile-time container ID

MATLAB you said ?

```
R = I(:,:,1);
G = I(:,:,2);
B = I(:,:,3);

Y = min(abs(0.299.*R+0.587.*G+0.114.*B),235);
U = min(abs(-0.169.*R-0.331.*G+0.5.*B),240);
V = min(abs(0.5.*R-0.419.*G-0.081.*B),240);
```

Now with NT^2

```
table<double> R = I(_,_,1);
table<double> G = I(_,_,2);
table<double> B = I(_,_,3);
table<double> Y, U, V;

Y = min(abs(0.299*R+0.587*G+0.114*B),235);
U = min(abs(-0.169*R-0.331*G+0.5*B),240);
V = min(abs(0.5*R-0.419*G-0.081*B),240);
```

Now with NT^2

```
table<double,settings(shallow)> R = I(_,_,1);
table<double,settings(shallow)> G = I(_,_,2);
table<double,settings(shallow)> B = I(_,_,3);
table<double> Y, U, V;

Y = min(abs(0.299*R+0.587*G+0.114*B),235);
U = min(abs(-0.169*R-0.331*G+0.5*B),240);
V = min(abs(0.5*R-0.419*G-0.081*B),240);
```

Now with NT^2

```
table<float,settings(shallow,of_size_<640,480>)> R = I(_,_,1);
table<float,settings(shallow,of_size_<640,480>)> G = I(_,_,2);
table<float,settings(shallow,of_size_<640,480>)> B = I(_,_,3);
table<float,settings(of_size_<640,480>)> Y, U, V;

Y = min(abs(0.299*R+0.587*G+0.114*B),235);
U = min(abs(-0.169*R-0.331*G+0.5*B),240);
V = min(abs(0.5*R-0.419*G-0.081*B),240);
```

Evaluation Contexts

Use case

- Some settings are expression-based, not array-based
- Some expressions need hints to be compiled properly
- Some expressions want to be stored via type-erasure

The `context` class

```
table<float,settings(shallow,of_size_<640,480>)> R = I(_,_,1);
table<float,settings(shallow,of_size_<640,480>)> G = I(_,_,2);
table<float,settings(shallow,of_size_<640,480>)> B = I(_,_,3);
table<float,settings(of_size_<640,480>)> Y, U, V;

context< unroll<5> >()
[
  Y = min(abs(0.299*R+0.587*G+0.114*B),235),
  U = min(abs(-0.169*R-0.331*G+0.5*B),240) ,
  V = min(abs(0.5*R-0.419*G-0.081*B),240)
];
```

Evaluation Contexts

Use case

- Some settings are expression-based, not array-based
- Some expressions need hints to be compiled properly
- Some expressions want to be stored via type-erasure

The `context` class

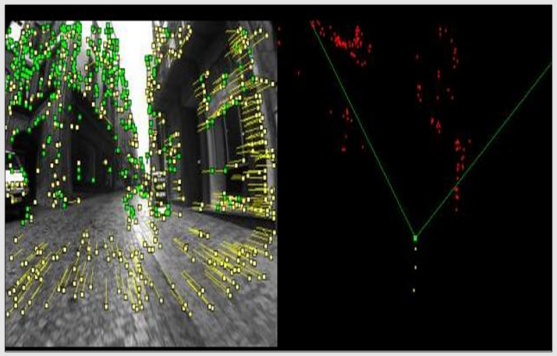
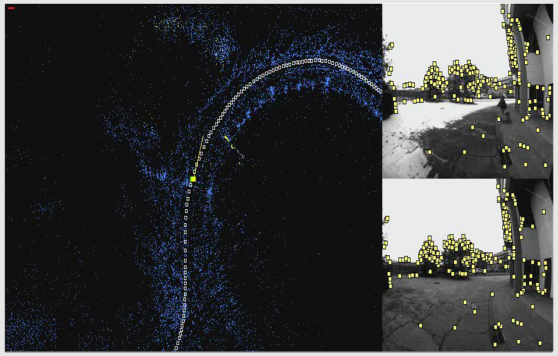
```
table<float,settings(shallow,of_size_<640,480>)> R = I(_,_,1);
table<float,settings(shallow,of_size_<640,480>)> G = I(_,_,2);
table<float,settings(shallow,of_size_<640,480>)> B = I(_,_,3);
table<float,settings(of_size_<640,480>)> Y, U, V;

// Compilation and storage in a function object
function<void()> k;

k = context< unroll<5> >()
[
  Y = min(abs(0.299*R+0.587*G+0.114*B),235),
  U = min(abs(-0.169*R-0.331*G+0.5*B),240) ,
  V = min(abs(0.5*R-0.419*G-0.081*B),240)
];

k();
```

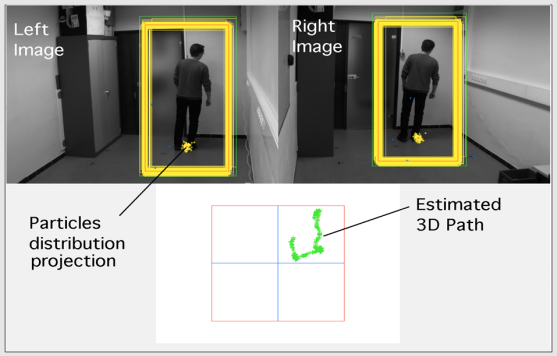
Some Performances

RGB2YUV timing (in cycles/pixels)

Size	128x128	256x256	512x512	1024x1024
MATLAB 2010a (2 cores)	85	89	97	102
C (1 core)	23.6	23.8	23.9	24.0
NT^2 (1 core)	22.3	22.4	22.2	24.1
NT^2 OpenMP (2 cores)	11.4	11.4	11.5	12.2
NT^2 SIMD	5.5	5.6	5.6	5.9
NT^2 SIMD+OpenMP	2.9	2.9	2.9	3.0
NT^2 SIMD speed-up	3.9	3.9	3.9	4.0
NT^2 OpenMP speed-up	1.92	1.96	1.98	1.95
NT^2 vs MATLAB speed-up	29.3	30.7	33.5	34

Some real life applications

Simultaneous Localization And Mapping	Convoi de véhicules autonomes	Navigation autonome par mémoire visuelle
		

Suivi par filtrage particulaire temps réel	Stabilisation d'images
	

Talk Layout

Embedded Domain Specific Languages

What's an EDSL ?

- DSL = Domain Specific Language
- Declarative language, easy-to-use, fitting the domain
- EDSL = DSL within a general purpose language

EDSL in C++

- Relies on operator overload abuse (Expression Templates)
- Carry semantic information around code fragment
- Generic implementation become self-aware of optimizations

Boost.Proto

What's Boost.Proto

- EDSL for defining EDSLs in C++
- Generalize Expression Templates
- Easy way to define and test EDSL

Boost.Proto Benefits

- Fast development process
- Compiler guys are at ease : Grammar + Semantic + Code Generation process
- Easily extendable through Transforms
- Easy to handle : MPL and Fusion compatible

Boost.Proto in NT^2

Objectives

- Scalability: Keep compile-time and code size low
- Extensibility: Simplify new optimizations design
- Debugging: Allow for meaningful error reporting

Solutions

- Provide a generic "compilation" transform
- Select the best function definition process
- Allow for non-intrusive new pass definition
- Integrate SFINAE and static assert as soon as possible

Generic EDSL Compilation

How does it work ?

- Perform a depth-first tree traversal on the AST
- Evaluate each terminal unless specified otherwise
- Turn each node into a Callable Function Object
- Apply said CFO to list of terminals
- Pass to parent node and repeat

Compile with `Boost::Proto` Transform

- Use `unpack` to turn a AST into a fusion sequence
- Add `apply< Visitor,tag_of<Xpr> >` to this sequence
- Turn it back into a function call with `proto::default_`
- Apply recursively

The Generic `compile` Transform

Implementation

```
template<class Visitor>
struct compile
     : or_< when< terminal< typename apply< Visitor, _ >::type >
                  , _value
                  >
          , when<_,traversal<Visitor>()>
          >
{};
```

Sample Use

```
template<class Xpr> template<class T, class Opts>
table<T,Opts>& operator=( expression<Xpr, container<table_> > const& xpr )
{
  resize( compile< size<_> >()(xpr) ); // or size(xpr)

  // Simplified evaluation process
  for(std::size_t i=0;i<size();++i)
   value(*this)(i) = compile< functor<_> >()(xpr,i,i);
}
```

The Generic `compile` Transform

`traversal`

- Extension point for node traversal
- Can be extended directly or through `Visitor`
- Can be extended by via `meta::process_node`

The Generic `compile` Transform

`traversal`

- Extension point for node traversal
- Can be extended directly or through `Visitor`
- Can be extended by via `meta::process_node`

`process_node`

- Extension point for node processing
- Can be built from a selection of classic tree traversal strategies
- Can be overloaded partially or completely

traversal

Implementation

```
template<class Visitor>
struct traversal : transform< traversal<Visitor> >
{
  template<class X, class S, class D>
  struct impl : transform_impl<X,S,D>
  {
    typedef typename tag_of<X>::type    tag;
    typedef typename
    boost::result_of< meta::process_node<Visitor,tag> ( typename impl::expr
                                                      , typename impl::state
                                                      , typename impl::data
                                                      )
                    >::type                           result_type;

    inline result_type  operator()( typename impl::expr_param   x
                                  , typename impl::state_param  s
                                  , typename impl::data_param   d
                                  ) const
    {
      return meta::process_node<Visitor,tag>()(x,s,d);
    }
  };
};
```

process_node

Implementation

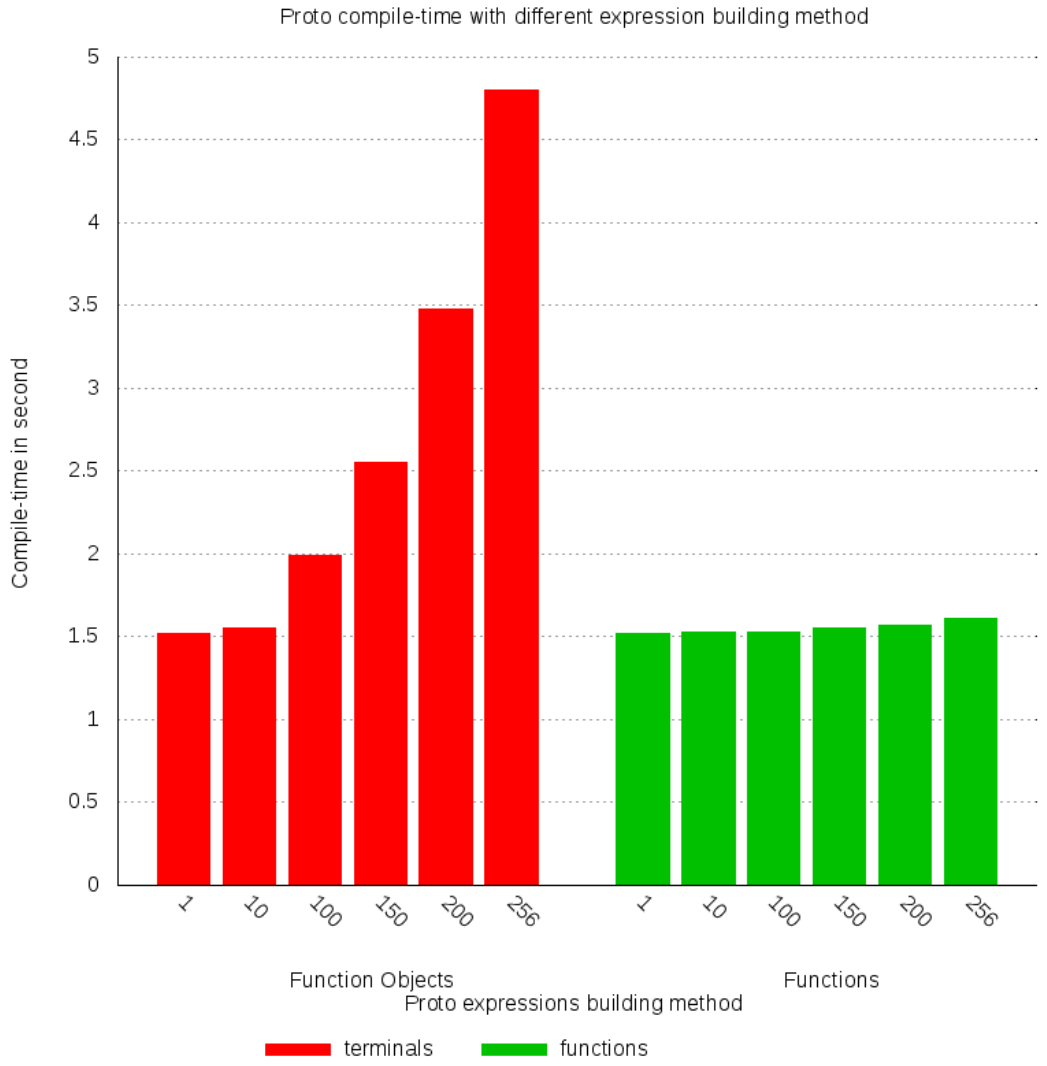
```
template<class Visitor,class Tag> struct process_node
{
  template<class Sig> struct result;
  template<class This,class X,class S, class D>
  struct  result<This(X,S,D)>
        : result_of<meta::recursive_process<Visitor,Tag>(X,S,D)> {};

  template<class X, class S, class D> inline
  typename result<process_node(X,S,D)>::type
  operator()(X const& x, S const& s, D const& d) const
  {
    return meta::recursive_process<Visitor,Tag>()(x,s,d);
  }
};
```

recursive_process

Implementation

```
template<class Visitor, class Tag> struct recursive_process
{
  typedef typename apply<Visitor,Tag>::type             visit;
  typedef typename terminal<visit>::type                term;
  typedef functionnal::unpack_expr<functors::function_> upck;
  typedef _default< meta::compile<Visitor> >            descent;

  template<class X> struct unpacker
  {
    typedef typename boost::result_of<meta::push_front(X,term)>::type xpr;
    typedef typename boost::result_of<upck(xpr)>::type                type;
  };

  template<class Sig> struct result;
  template<class This,class X,class S,class D>
  struct  result<This(X,S,D)>
        : boost::result_of<descent(typename unpacker<X>::type,S,D)>
  {};

  template<class X,class S,class D>
  inline typename result<recursive_process(X,S,D)>::type
  operator()( X const& x, S const& s, D const& d ) const
  {
    return descent()(upck()(meta::push_front()(x,term())),s,d);
  }
};
```

Impact on compile time



Talk Layout

Parallelism in NT^2

What's available ?

- Intra-processor SIMD instructions set
- Multi-core and multi-processors
- Clusters, Grid and other large-scale systems

What does NT^2 provide ?

- Compatible with `Boost.MPI`
- Support for `openMP` in regular cases
- Support mainstream SIMD extensions

Parallelism in NT^2

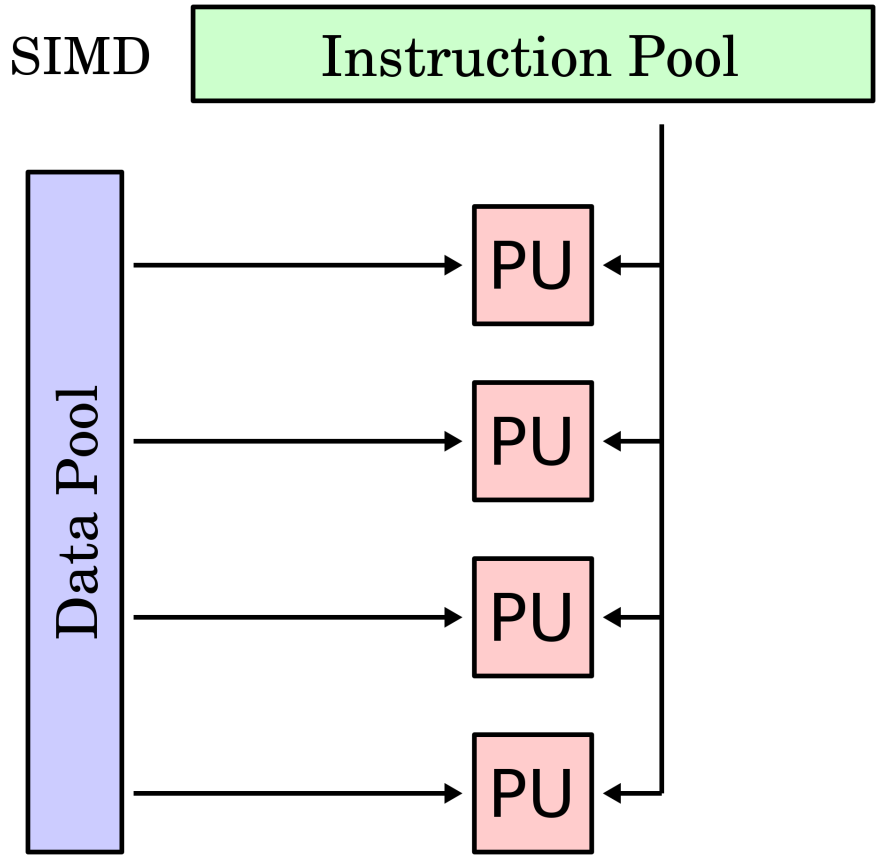
What's available ?

- Intra-processor SIMD instructions set
- Multi-core and multi-processors
- Clusters, Grid and other large-scale systems

What does NT^2 provide ?

- Compatible with `Boost.MPI`
- Support for `openMP` in regular cases
- Support mainstream SIMD extensions

SIMD in a nutshell



SIMD in a nutshell

Principles

- SIMD processor provides a special SIMD register file
- SIMD registers have a fixed size (64, 128, 256 bits)
- SIMD registers contains elements from a single type
- **cardinal** of a register is equal to the number of element in a register
- Irregular architecture dependant instructions sets
- Memory access with SIMD registers need to be aligned
- Not all operations are available for all types

SIMD Flavors

X86/IA64 architectures

- MMX provides 64 bits SIMD registers
- SSE2 to SSE4 provides 128 bits registers
- AVX mixes 128/256 bits registers
- All were designed around multimedia then fitted back

SIMD Flavors

X86/IA64 architectures

- MMX provides 64 bits SIMD registers
- SSE2 to SSE4 provides 128 bits registers
- AVX mixes 128/256 bits registers
- All were designed around multimedia then fitted back

PowerPC architectures

- PPC Altivec provides 128 bits registers (but no double)
- Cell Altivec provides 128 bits registers (but only for float and short)
- General Purpose API with few dedicated instructions

SIMD Flavors

X86/IA64 architectures

- MMX provides 64 bits SIMD registers
- SSE2 to SSE4 provides 128 bits registers
- AVX mixes 128/256 bits registers
- All were designed around multimedia then fitted back

PowerPC architectures

- PPC Altivec provides 128 bits registers (but no double)
- Cell Altivec provides 128 bits registers (but only for float and short)
- General Purpose API with few dedicated instructions

Other architectures

- ARM Cortex NEON with 64 bits registers
- DSP oriented features

SIMD in practice

What SIMD should bring?

- Speed-up equals to vector's cardinal
- Usage of more registers
- Potential Fused operation (madd, dot,...)

SIMD in practice

What SIMD should bring?

- Speed-up equals to vector's cardinal
- Usage of more registers
- Potential Fused operation (madd, dot,...)

What do we get ?

- In SP, speed-up are around 4 and sometimes super-linear
- In DP, speed-up around 1.5 and few to none super-linear effects
- For integers, speed-up is around 50 to 80% of the maximum speed-up
- For integers, a lot of features are missing and need to be mapped.

Toward a Generic Solution

Objectives

- Provide portable SIMD related tools
- Hide architecture differences
- Provide a nice fallback
- Take cares of details

Toward a Generic Solution

Objectives

- Provide portable SIMD related tools
- Hide architecture differences
- Provide a nice fallback
- Take cares of details

Solutions

- Encapsulate the Concept of SIMD register
- Provide aligned allocator
- Provide iterator for SIMD chunking

Toward a Generic Solution

`simd::register_<T,N>`

- Packs `N` values of type `T`
- If (`T`,`N`) maps onto a native SIMD type, we use it else we fallback to `boost::array<T,N>`
- Detection and replacement of fused operations
- `simd::register_<T,N>` is an immutable Fusion Sequence

Toward a Generic Solution

`simd::register_<T,N>`

- Packs `N` values of type `T`
- If (`T`,`N`) maps onto a native SIMD type, we use it else we fallback to `boost::array<T,N>`
- Detection and replacement of fused operations
- `simd::register_<T,N>` is an immutable Fusion Sequence

Related tools

- `simd::allocator<T>` allocates `T` in a aligned way
- Allocators adaptor are provided to build aligned allocators with specific behavior
- `simd::iterator<T,N>` iterates over container of `T` and returns `register_<T,N>`
- Meta-functions: `is_vectorizable`, `scalar_of`, `cardinal_of`

Toward a Generic Solution

Sample usage

```
int main()
{
  std::vector<float, simd::allocator<float> > v(1000);

  simd::register_<float,4> c(1,2,3,4);
  simd::iterator<float,4> b = simd::begin(v);
  simd::iterator<float,4> e = simd::end(v);

  while(b != e)
  {
    store(b, *b + cos(*b/c));
    b++;
  }
}
```

Integration in NT^2

From the user point of view

- Vectorization of **NT^2** expressions is automatic
- **NT^2** containers take care of aligned memory allocation
- Partial support for **Struct of Array** $\leftrightarrows$ **Array of Struct** conversion

Integration in NT^2

From the user point of view

- Vectorization of **NT^2** expressions is automatic
- **NT^2** containers take care of aligned memory allocation
- Partial support for **Struct of Array** $\leftrightarrows$ **Array of Struct** conversion

From the developer point of view

- `simd::register_` is a stand-alone component
- Simplifies design of SIMD enabled computations
- Can be easily extended for new architectures

SIMD Pitfalls

What will be ideal?

- At least support the equivalent of `cmath`
- ... With a proper accuracy on the full range
- ... With good performances
- ... On all sensible types

SIMD Pitfalls

What will be ideal?

- At least support the equivalent of `cmath`
- ... With a proper accuracy on the full range
- ... With good performances
- ... On all sensible types

Where are the limiting factors?

- Accuracy problems
- Type conversions are clunky
- Too much branching is deadly

SIMD Do and Don't

Floating numbers accuracy

- A 128 bits vector contains 4 floats of 32 bits and every bits are used
- Scalar FPU works internally on an extended 80 bits representation
- Precision is greater in scalar computations than in SIMD
- But depends of registers use and optimization levels

SIMD Do and Don't

Floating numbers accuracy

- A 128 bits vector contains 4 floats of 32 bits and every bits are used
- Scalar FPU works internally on an extended 80 bits representation
- Precision is greater in scalar computations than in SIMD
- But depends of registers use and optimization levels

Playing with type casting

- SIMD intrinsics generally do not mix vector types
- Two ways to cast vectors:
  - C style cast: 0 cycles, reinterpret vector bits
  - Conversion cast using intrinsics: 2-3 cycles but modify data
- Useful conversions limited to: float/double $\leftrightarrow$ int32_t/int64_t
- C style cast enables fast bitwise operations on all types and allows masking

SIMD Do and Don't

Too much branching is deadly

- If a scalar algorithm branches, it takes the right path
- If a SIMD algorithm branches, it evaluates all alternatives and select the results in parallel.
- No branching on vector of 2 elements as $2 \times \frac{1}{2} = 1$

SIMD Do and Don't

Too much branching is deadly

- If a scalar algorithm branches, it takes the right path
- If a SIMD algorithm branches, it evaluates all alternatives and select the results in parallel.
- No branching on vector of 2 elements as $2 \times \frac{1}{2} = 1$

Getting Branching Back

- Array may contain (partially) ordered values
- SIMD vectors loaded from such arrays may gather elements using the same heavy branch of an algorithm.
- One can use horizontal branching with scalar tests.

SIMD Do and Don't

Horizontal Scalar Branching

- Scalar test using a SIMD reduction function
- SIMD reduction takes a vector and return a scalar
- Examples: `all`, `any`, `sum`.

SIMD Do and Don't

Horizontal Scalar Branching

- Scalar test using a SIMD reduction function
- SIMD reduction takes a vector and return a scalar
- Examples: `all`, `any`, `sum`.

HSB with one heavy branch

```
if all(v>v0)   r = algo1(); // algo2 is heavy
else           r = select (v>v0,algo1(),algo2());
```

SIMD Do and Don't

Horizontal Scalar Branching

- Scalar test using a SIMD reduction function
- SIMD reduction takes a vector and return a scalar
- Examples: `all`, `any`, `sum`.

HSB with two heavy branches

```
if all(v>v0)          r = algo1();
else if all(v<=v0)    r = algo2();
else                  r = select (v>v0,algo1(),algo2);
```

SIMD Function Design Rationale

The Historical Process

- We called standard functions
- We used free libraries which inspired our SIMD version of those
- SIMD algorithms were then 'scalarized' back

What dis we use?

- `crlibm` was used for precision checking
- `libc`, `cephes`, `fdlibm` and `Boost.Math` for inspiration
- and all are provided in **NT^2** as adapted toolboxes in proper namespaces

Use-case Trigonometric Functions

Common algorithm sketch:

1. Reduce the computational range
2. Use a polynomial approximation on the reduced range
3. Finalize if needed

Use-case Trigonometric Functions

Common algorithm sketch:

1. Reduce the computational range
2. Use a polynomial approximation on the reduced range
3. Finalize if needed

Advantage:

Three different functions for the price of one, by specializing steps 1 and 3.

- $\cos(x)$ cosine of x in radian
- $cosd(x)$ cosine of x in degree
- $cospi(x)$ cosine of $\pi \times x$

Input Range Reduction

Problem

- Reduction can be very expansive in the standard case
- We need to compute the integer quotient of $|x|$ by $\frac{\pi}{4}$ and the floating remainder of this division
- The difficulty lies in the facts that:
  - π is not exactly representable
  - the results must be kept with maximum precision, which can be difficult if x is near a big integer multiple of π

Input Range Reduction

Problem

- Reduction can be very expansive in the standard case
- We need to compute the integer quotient of $|x|$ by $\frac{\pi}{4}$ and the floating remainder of this division
- The difficulty lies in the facts that:
  - π is not exactly representable
  - the results must be kept with maximum precision, which can be difficult if x is near a big integer multiple of π

Speed *vs* Accuracy

The range has to be reduced to $[0, \frac{\pi}{4}]$, keeping quadrant information

- $\forall x, |x| \leq \frac{\pi}{4} \Rightarrow$ Nothing to do $\rightarrow$ fast version
- $\forall x, |x| \leq 20\pi \Rightarrow$ Reduction is as costly as an evaluation $\rightarrow$ average version
- else preserving accuracy is too slow $\rightarrow$ slow version or accuracy loss

SIMD Evaluation of cosine

Evaluation Process

- If the range is not contained in $[-\frac{\pi}{4}, \frac{\pi}{4}]$, sine and cosine must be both evaluated because of the reduction scheme
- We choose if it is sine or cosine value to return
- If it was a special case, we return `NaN`

Basic Evaluation ($|x| \leq \frac{\pi}{4}$)

- Polynomial with Horner or Estrin Scheme to minimize evaluation complexity
- Degrees are around 4 for float 8 for double

SIMD Evaluation of cosine

Putting everything together

```
static inline A0 cos_eval(const A0& z)
{
  const A0 y=horner<NT2_HORNER_COEFF_T(A0,3,(0x37ccf5ce,0xbab60619 0x3d2aaaa5))>(z);
  return oneplus(fma(z,Mhalf<A0>(),y*sqr(z)));
}
```

SIMD Evaluation of cosine

Putting everything together

```
template <class A0> A0 cosa(A0 a0)
{
  typedef typename as_integer<A0>::type i_t;
  const int de = sizeof(typename scalar_of<i_t>::type)*8-1;
  A0 x = abs(a0);
  if(any(replacement_needed(x)))
  {
    A0 y;
    map(cos_replacement, a0, y);
    return y;
  }
  else
  {
    A0 xr = Zero<A0>();
    i_t n = reduce(x, xr);
    i_t swp_bit = b_and(n, One<i_t>());
    i_t sgn_bit = shli(b_xor(swp_bit,shri(b_and(n, Two<i_t>()),1)),de);
    A0 z = sqr(xr);
    z = b_xor( select(isnez(swp_bit), sin_eval(z), cos_eval(z) )
             , sgn_bit
             );
    return b_or(isinvalid(a0),z);
  }
}
```

Performances out of the box for float

Timing performance (in cycles/value)

			scalar	simd	scalar	simd
range	std	crm(rn)	nt2 fast	nt2 fast	nt2	nt2
$[0, \frac{\pi}{4}]$	108	135	16	3	23	10
$[0, 20\pi]$	109	173	n/a	n/a	49	17
$[0, 10^4]$	110	187	n/a	n/a	122	122
$[0, 10^{30}]$	288	2013	n/a	n/a	275	275

Accuracy against `crlibm` (in ulp)

range	std	nt2 fast	nt2
$[0, \frac{\pi}{4}]$	0	0.5	0.5
$[0, 20\pi]$	0	n/a	0.5
$[0, 10^4]$	0	n/a	0.5
$[0, 10^{30}]$	10^7	n/a	10^7

Performances out of the box for float

Timing performance (in cycles/value)

function	range	std or other	crm(rn)	scalar nt2	simd nt2
exp	$[-10, 10]$	160	124	83	13
log	$[0, 10^4]$	100	82	73	13
expm1	$[-10, 10]$	n/a	176	110	39
log1p	$[-1, 10^4]$	n/a	204	87	14
acos	$[-1, 1]$	234	285	92	32
asin	$[-1, 1]$	226	275	67	10
floor	$[-10^4, 10^4]$	55	n/a	36	2
ellpe	$[0, 1]$	373(*)	n/a	174	24

(*) cephes

Talk Layout

Support for UDTs

User-Defined Types

- Domain specific application may require specific abstract type
- May make sense to have table or matrix of those
- Supporting UDTs easily is a key extension point

Support for UDTs

User-Defined Types

- Domain specific application may require specific abstract type
- May make sense to have table or matrix of those
- Supporting UDTs easily is a key extension point

Solution

- Non-intrusive type registration
- Reuse whatever operators that already exist
- Provide incremental support definition

Tag Dispatching Strategy

Rank, Granularity and Dominant Types

- Every pair of types is ordered via its Granularity/Rank value.
- **Granularity:** is the type a scalar or a container ?
- **Rank:** relative position among all types of same **Granularity**.
- A non-registered type has a Granularity/Rank of (∞, ∞).

Type Discrimination

- For any values of types $T_0, \ldots, T_n$, we compute the **Dominant** type, (i.e. T_i with the largest granularity and rank)
- This type's category is then used to select the functor implementation

Support for UDTs

External Registration

```
#include <boost/math/quaternion.hpp>
#include <nt2/sdk/functor/preprocessor/category.hpp>

NT2_REGISTER_CATEGORY(quaternion_,5000);
NT2_SET_CATEGORY_TPL(1,boost::quaternion, scalar_<quaternion_>)
```

Internal Registration

```
NT2_REGISTER_CATEGORY(quaternion_,5000);

template<class T> class quaternion
{
  typedef scalar_<quaternion_>  category_tag;
  // ...
};
```

Adding New Functions

What do we have then?

NT^2 has more than 250 math functions callable on almost anything

Objectives

- How to control supported types in function calls ?
- How to emit graceful error messages ?
- How to easily add more functions or more specializations ?

Handling Functions Definition

Limitations of classic solutions

- Manually enumerating overloads is a quadratic process
- Partial template overload on arguments may lead to ambiguities
- How to handle new, user-defined types ?

Solution: Tag-Dispatched Callable Object

```
struct sqrt_;
```

Handling Functions Definition

Limitations of classic solutions

- Manually enumerating overloads is a quadratic process
- Partial template overload on arguments may lead to ambiguities
- How to handle new, user-defined types ?

Solution: Tag-Dispatched Callable Object

```
template<class A0>
typename boost::result_of<functor<sqrt_>(A0)>::type
sqrt( A0 const& a0 )
{
  functor<sqrt_> callee;
  return callee(a0);
}
```

Specializing Functions Behavior

Our solution

- Function implementation are done in the `call` structure
- `call` is internally used by `functor`
- `call` is specialized per function tag and/or type category

Example

```
namespace nt2 { namespace functors
{
  template<class FunctionTag,class Category,class Dummy=void>
  struct call
  {
    template<class Sig> struct result;

    template<class ...Args>
    typename result<call(Args...)>::type
    operator()( Args... const& ) const;
  };
} }
```

Specializing Functions Behavior

Our solution

- Function implementation are done in the `call` structure
- `call` is internally used by `functor`
- `call` is specialized per function tag and/or type category

Example

```
namespace nt2 { namespace functors
{
  template<class Dummy,class T>
  struct call<sqrt_, scalar<T>, Dummy>
  {
    template<class Sig> struct result;
    template<class This,class A>
    struct result<This(A)> { typedef typename meta::as_real<A>::type type };

    NT2_FUNCTOR_CALL(1) { return std::sqrt(a0); }
  };
} }
```

Tag Dispatching Strategy (again)

Handling Polymorphism

- All NT^2 functions are inherently polymorphic
- We need to discriminate calls to select the proper implementation
- Two level **Tag-Dispatching**:
  - Discriminate type family at call site
  - Discriminate type in implementation variants

Type Discrimination

```
template<class This,class ...Args> struct result<This(...Args)>
{
  typedef typename meta::dominant<Args...>::type                  cat;
  typedef typename boost::result_of<call<Tag,cat>(Args...)>::type type;
};

template<class ...Args> inline result<functor(Args...)>::type
operator()( Args&&... args ) const
{
  typedef typename meta::dominant<Args...>::type cat;
  return call<Tag,cat>()(std::forward<Args>(args)...);
}
```

Restricting Function Domain

SFINAE for everyone

- As for `call`, `validate` is an overloadable structure used in the SFINAE support of function calls.
- It allows fine tuning of which types or family of types are permitted.
- It can be specialized on tag or category or both.
- **Special Bonus**: in some case, it's not even needed, `call` can do it.

Example

```
namespace nt2 { namespace functors
{
  template<class Dummy,class T>
  struct validate< sqrt_, scalar<T>, Dummy >
  {
    template<class Sig> struct result;
    template<class This,class A0>
    struct result : boost::is_floating_points<A0> {};
  };
} }
```

Restricting Function Domain

SFINAE for everyone

- As for `call`, `validate` is an overloadable structure used in the SFINAE support of function calls.
- It allows fine tuning of which types or family of types are permitted.
- It can be specialized on tag or category or both.
- **Special Bonus**: in some case, it's not even needed, `call` can do it.

Example

```
template<class A0>
typename boost::lazy_enable_if<
       boost::result_of<typename functor<sqrt_>::validate(A0)>::type
     , boost::result_of<functor<sqrt_>(A0)>
                        >::type
sqrt( A0 const& a0 )
{
  functor<sqrt_> callee;
  return callee(a0);
}
```

Restricting Function Domain

SFINAE for everyone

- As for `call`, `validate` is an overloadable structure used in the SFINAE support of function calls.
- It allows fine tuning of which types or family of types are permitted.
- It can be specialized on tag or category or both.
- **Special Bonus**: in some case, it's not even needed, `call` can do it.

Example

```
template<class T>
struct call< plus_, simd<T> >
{
  template<class Sig> struct result;
  template<class This,class A>
  struct result<This(A,A)> { typedef A type; };

  NT2_FUNCTION_CALL(2) { return vec_add(a0,a1); }
};
```

Bringing Everything Together

Implementation details

- Specialize `call` and `validate` for unknown types so calls to unsupported functions or types fails gracefully
- Provide macros to generate all these by the user
- **Bonus:** All NT^2 functions are already Phoenix-compatible

Example

```
template<class Tag> struct call<Tag,unknown_>
{
  template<class Sig> struct result;
  NT2_STATIC_ASSERT_MSG(UNSUPPORTED_FUNCTION_CALL);
};

template<class Tag> struct validate<Tag,unknown_>
{
  typedef boost::mpl::false_ result_type;
};
```

Bringing Everything Together

Implementation details

- Specialize `call` and `validate` for unknown types so calls to unsupported functions or types fails gracefully
- Provide macros to generate all these by the user
- **Bonus:** All NT^2 functions are already Phoenix-compatible

Example

```
int main()
{
  int k= 4;
  cout << nt2::sqrt(k) << endl;
  cout << nt2::sqrt("lol") << endl;
}
```

Bringing Everything Together

Implementation details

- Specialize `call` and `validate` for unknown types so calls to unsupported functions or types fails gracefully
- Provide macros to generate all these by the user
- **Bonus:** All NT^2 functions are already Phoenix-compatible

Example

```
line 3: error: No call to function 'nt2::sqrt(int)'
...
line 4: error: static_assert::(****UNSUPPORTED_FUNCTION_CALL****)(nt2::functors::
     sqrt_(std::string))
```

Bringing Everything Together

Implementation details

- Specialize `call` and `validate` for unknown types so calls to unsupported functions or types fails gracefully
- Provide macros to generate all these by the user
- **Bonus:** All NT^2 functions are already Phoenix-compatible

Example

```
namespace nt2
{
  namespace functors
  {
    struct abs_ {};
  }

  NT2_FUNCTION_IMPLEMENTATION( functors::abs_, abs, 1)
}

#include NT2_INCLUDE(abs.hpp)
```

Bringing Everything Together

Implementation details

- Specialize `call` and `validate` for unknown types so calls to unsupported functions or types fails gracefully
- Provide macros to generate all these by the user
- **Bonus:** All NT^2 functions are already Phoenix-compatible

Example

```
template<class Dummy> struct call<abs_,scalar_<native_>,Dummy>
{
  template<class Sig> struct result;
  template<class This,class A0> struct result<This(A0)> : boost::result_of<meta::
       arithmetic(A0)>{};

  NT2_FUNCTOR_CALL_DISPATCH( 1
                           , typename nt2::meta::scalar_of<A0>::type
                           , (3, (real_,unsigned_,signed_))
                           )

  NT2_FUNCTOR_CALL_EVAL_IF(1, unsigned_) { return a0;              }
  NT2_FUNCTOR_CALL_EVAL_IF(1, signed_)   { return (a0 > 0)?a0:-a0; }
  NT2_FUNCTOR_CALL_EVAL_IF(1, real_)     { return std::abs(a0);    }
};
```

Bringing Everything Together

Implementation details

- Specialize `call` and `validate` for unknown types so calls to unsupported functions or types fails gracefully
- Provide macros to generate all these by the user
- **Bonus:** All NT^2 functions are already Phoenix-compatible

Example

```
#include <nt2/include/lambda/sqrt.hpp>

using nt2::lambda::sqrt;
using boost::phoenix::args::_1;


std::transform(v.begin(),v.end(), 1 + sqrt(_1) );
```

Talk Layout

Let's round this up!

Computing for Scientist

- If **NT^2** is not perfect, it is designed to be extendable w/r to users demands and needs
- Contrary to other array/algebra library, **NT^2** looks strange for C++ users and easy for MATLAB users
- `Boost` simplifies platform support and modularization
- `Boost::Proto` helps us writing our code as real EBNF and semantic actions

Current and Future Works

What we're cooking at the moment

- GPU support using `context`
- Sparse matrix support
- MPI support: PhD starting in 2010

Current and Future Works

What we're cooking at the moment

- GPU support using `context`
- Sparse matrix support
- MPI support: PhD starting in 2010

What we'll be cooking in the future

- Unify optimization process thanks to the **polyhedral model**
- Support for Embedded System (Cell, ARM, etc)
- Multi-stage programming for exotic platform
- Toward a global generic approach to parallelism ?

Giving back to `Boost`

How about `Boost.SIMD` ?

- NT^2 SIMD support is stand-alone and can be extracted
- Generalized functor dispatch system too
- Free bonus: fast scalar math function implementations
- Need help for SIMD implementation of `Boost.Math`

Giving back to Boost

How about Boost.SIMD ?

- NT^2 SIMD support is stand-alone and can be extracted
- Generalized functor dispatch system too
- Free bonus: fast scalar math function implementations
- Need help for SIMD implementation of Boost.Math

From NT^2 to Phoenix v3

- NT^2 generic AST evaluator is stand alone too
- Part of the Phoenix v3 prototype submitted for GSoC
- Turn it into a first-class transform in Boost.Proto ?

Thanks for your attention