Context

In Scientific Computing ... 

- there is Scientific
  - Applications are domain driven
  - Users ≠ 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 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
- Helps compilers to generate better parallel code
- Increase sustainability by decreasing amount of code to write

The Means

- Parallel Abstractions: Skeletons
- Efficient Implementation: DSEL
- The Holy Glue: Generative Programming
Efficient or Expressive – Choose one

- Matlab
- SciLab
- C++
- JAVA
- C, FORTRAN

Expressivité vs. Efficacité
Talk Layout

Introduction

Abstractions

Efficiency

Tools

Conclusion
Talk Layout

Introduction

Abstractions

Efficiency

Tools

Conclusion
Spotting abstraction when you see one
Spotting abstraction when you see one
Parallel Skeletons in a nutshell

Basic Principles [COLE 89]

- There are patterns in parallel applications
- Those patterns can be generalized in *Skeletons*
- Applications are assembled as combination of such patterns
Parallel Skeletons in a nutshell

Basic Principles [COLE 89]

- There are patterns in parallel applications
- Those patterns can be generalized in *Skeletons*
- Applications are assembled as combination of such patterns

Functionnal point of view

- Skeletons are *Higher-Order Functions*
- Skeletons support a compositionnal semantic
- Applications become composition of state-less functions
Classic Parallel Skeletons

Data Parallel Skeletons

- **map**: Apply a n-ary function in SIMD mode over subset of data
- **fold**: Perform n-ary reduction over subset of data
- **scan**: Perform n-ary prefix reduction over subset of data
Classic Parallel Skeletons

Data Parallel Skeletons

- **map**: Apply a n-ary function in SIMD mode over subset of data
- **fold**: Perform n-ary reduction over subset of data
- **scan**: Perform n-ary prefix reduction over subset of data

Task Parallel Skeletons

- **par**: Independant task execution
- **pipe**: Task dependency over time
- **farm**: Load-balancing
Why using Parallel Skeletons

Software Abstraction

- Write without bothering with parallel details
- Code is scalable and easy to maintain
- Debuggable, Provable, Certifiable
Why using Parallel Skeletons

Software Abstraction
- Write without bothering with parallel details
- Code is scalable and easy to maintain
- Debuggable, Provable, Certifiable

Hardware Abstraction
- Semantic is set, implementation is free
- Composability $\Rightarrow$ Hierarchical architecture
Talk Layout

Introduction

Abstractions

Efficiency

Tools

Conclusion
Generative Programming as a Tool

Available techniques

- Dedicated compilers
- External pre-processing tools
- Languages supporting meta-programming
Generative Programming as a Tool

Available techniques

- Dedicated compilers
- External pre-processing tools
- Languages supporting meta-programming
Generative Programming as a Tool

Available techniques

- Dedicated compilers
- External pre-processing tools
- Languages supporting meta-programming

Definition of Meta-programming
Meta-programming is the writing of computer programs that analyse, transform and generate other programs (or themselves) as their data.
From Generative to Meta-programming

Meta-programmable languages

- TEMPLATE HASKELL
- metaOcaml
- C++
From Generative to Meta-programming

Meta-programmable languages

- **TEMPLATE HASKELL**
- **metaOcaml**
- **C++**
From Generative to Meta-programming

Meta-programmable languages

- TEMPLATE HASKELL
- metaOcaml
- C++

C++ meta-programming

- Relies on the C++ TEMPLATE sub-language
- Handles types and integral constants at compile-time
- Proved to be Turing-complete
Domain Specific Embedded Languages

What’s an DSEL ?

- DSL = Domain Specific Language
- Declarative language, easy-to-use, fitting the domain
- DSEL = 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

Exploiting static AST

- At the expression level: code generation
- At the function level: inter-procedural optimization
Expression Templates

```cpp
matrix x(h,w), a(h,w), b(h,w);
x = cos(a) + (b*a);

expr<assign , expr<matrix&> , expr<plus,
    expr<cos , expr<matrix&>> >
   , expr<multiplies , expr<matrix&>, expr<matrix&>> >
>(x, a, b);
```

```
#pragma omp parallel for
for(int j=0;j<h;++j)
{
    for(int i=0;i<w;++i)
    {
        x(j,i) = cos(a(j,i)) + (b(j,i) * a(j,i));
    }
}
```

Arbitrary Transforms applied on the meta-AST
Embedded Domain Specific Languages

EDSL in C++
- Relies on operator overload abuse
- Carry semantic information around code fragment
- Generic implementation become self-aware of optimizations

Advantages
- Allow introduction of DSLs without disrupting dev. chain
- Semantic defined as type informations means compile-time resolution
- Access to a large selection of runtime binding
Talk Layout

Introduction

Abstractions

Efficiency

Tools

Conclusion
Different Strokes

Objectives
- Apply DSEL generation techniques for different kind of hardware
- Demonstrate low cost of abstractions
- Demonstrate applicability of skeletons

Contributions
- Extend DEMRAL into AA-DEMRAL
- Boost.SIMD
- NT²
Architecture Aware Generative Programming
Boost.SIMD
Generic Programming for portable SIMDization

- SIMD computation C++ Library
  - Built with modern C++ style for modern C++ usage
  - Easy to extend
  - Easy to adapt to new CPUs
Boost.SIMD

Generic Programming for portable SIMDization

- SIMD computation C++ Library
  - Built with modern C++ style for modern C++ usage
  - Easy to extend
  - Easy to adapt to new CPUs

- Goals:
  - Integrate SIMD computations in standard C++
  - Make writing of SIMD code easier
  - Promotes Generic Programming
  - Provides performance out of the box
Boost.SIMD

Principles

- `pack<T,N>` encapsulates the best hardware register for $N$ elements of type $T$
- `pack<T,N>` provides a classical value semantic
- Operations on `pack<T,N>` maps to proper intrinsics
- Support for SIMD standard algorithms
Boost.SIMD

Principles

- `pack<T,N>` encapsulates the best hardware register for \( N \) elements of type \( T \)
- `pack<T,N>` provides a classical value semantic
- Operations on `pack<T,N>` maps to proper intrinsics
- Support for SIMD standard algorithms

How does it work?

- Code is written as for scalar values
- Code can be applied as polymorphic functor over data
- Expression Templates optimize operations
Boost.SIMD

Current Support

- OS: Linux, Windows, iOS, Android
- Architecture: x86, ARM, PowerPC
- Compilers: g++, clang, icc, MSVC
- Extensions:
  - x86: SSE2, SSE3, SSSE3, SSE4.x, AVX, XOP, FMA3/4, AVX2
  - PPC: VMX
  - ARM: NEON
- In progress:
  - Xeon Phi MIC
  - VSX, QPX
  - NEON2
pack<int> julia(pack<float> const& a, pack<float> const& b) {
    pack<int> i_t;
    std::size_t i = 0;
    pack<int> iter;
    float x, y;

    do {
        pack<float> x2 = x * x;
        pack<float> y2 = y * y;
        pack<float> xy = 2 * x * y;
        x = x2 - y2 + a;
        y = xy + b;
        pack<float> m2 = x2 + y2;
        iter = selinc(m2 < 4, iter);
        i++;
    } while (any(mask) && i < 256);

    return iter;
}
pack<int> julia(pack<float> const & a, pack<float> const & b)
{
    pack<int> i_t;
    std::size_t i = 0;
    pack<int> iter;
    float x, y;

    do
    {
        pack<float> x2 = x * x;
        pack<float> y2 = y * y;
        pack<float> xy = 2 * x * y;
        x = x2 - y2 + a;
        y = xy + b;
        pack<float> m2 = x2 + y2;  
        iter = selinc(m2 < 4, iter);
        i++;
    }
    while (any(mask) && i < 256);

    return iter;
}
pack<int> julia(pack<float> const& a, pack<float> const& b)
{
    pack<int> i_t;
    std::size_t i = 0;
    pack<int> iter;
    float x, y;

    do
    {
        pack<float> x2 = x * x;
        pack<float> y2 = y * y;
        pack<float> xy = 2 * x * y;
        x = x2 - y2 + a;
        y = xy + b;
        pack<float> m2 = x2 + y2;
        iter = selinc(m2 < 4, iter);
        i++;
    }
    while(any(mask) && i < 256);

    return iter;
}
NT²

A Scientific Computing Library

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

Components

- Use Boost.SIMD for in-core optimizations
- Use recursive parallel skeletons for threading
- Code is made independant of architecture and runtime
The Numerical Template Toolbox

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics `MATLAB` array behavior and functionalities
- 500+ functions usable directly either on `table` or on any scalar values as in `MATLAB`
The Numerical Template Toolbox

Principles

- \texttt{table<T,S>} is a simple, multidimensional array object that exactly mimics \texttt{MATLAB} array behavior and functionalities
- 500+ functions usable directly either on \texttt{table} or on any scalar values as in \texttt{MATLAB}

How does it work

- Take a \texttt{.m} file, copy to a \texttt{.cpp} file
The Numerical Template Toolbox

Principles

- `table<T,S>` is a simple, multidimensional array object that exactly mimics `Matlab` array behavior and functionalities.
- 500+ functions usable directly either on `table` or on any scalar values as in `Matlab`.

How does it work

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

Principles

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

How does it work

- 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`
NT2 - From MATLAB ...

A1 = 1:1000;
A2 = A1 + randn(size(A1));

X = lu(A1*A1');

rms = sqrt( sum(sqr(A1(:) - A2(:))) / numel(A1) );
table<

double> A1 = _(1.,1000.);

table<

double> A2 = A1 + randn(size(A1));


table<

double> X = lu( mtimes(A1, trans(A1) ));

double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
Performances - Mandelbrodt

![Mandelbrot Performance Graph]

- scalar
- sse
- sse2
- sse3
- sse4.2
- avx
- avx2
- no penmsg/
- avx2 nt2
- avx2 nt2

The graph shows performance metrics for different resolutions and vectorization techniques.
Performances - LU Decomposition

8000 × 8000 LU decomposition

Median GFLOPS vs Number of cores

- NT²
- Plasma
Performances - linsolve

\[
\text{nt2::tie}(x,r) = \text{linsolve}(A,b);
\]

<table>
<thead>
<tr>
<th>Scale</th>
<th>C LAPACK</th>
<th>C MAGMA</th>
<th>NT(^2) LAPACK</th>
<th>NT(^2) MAGMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024 × 1024</td>
<td>85.2</td>
<td>85.2</td>
<td>83.1</td>
<td>85.1</td>
</tr>
<tr>
<td>2048 × 2048</td>
<td>350.7</td>
<td>235.8</td>
<td>348.2</td>
<td>236.0</td>
</tr>
<tr>
<td>12000 × 1200</td>
<td>735.7</td>
<td>1299.0</td>
<td>734.1</td>
<td>1300.1</td>
</tr>
</tbody>
</table>
Talk Layout

Introduction

Abstractions

Efficiency

Tools

Conclusion
Let’s round this up!

Parallel Computing for Scientist

- Software Libraries built as Generic and Generative components can solve a large chunk of parallelism related problems while being easy to use.
- Like regular language, EDSL needs informations about the hardware system
- Integrating hardware descriptions as Generic components increases tools portability and re-targetability
Let’s round this up!

Parallel Computing for Scientist

- Software Libraries built as Generic and Generative components can solve a large chunk of parallelism related problems while being easy to use.
- Like regular language, EDSL needs informations about the hardware system
- Integrating hardware descriptions as Generic components increases tools portability and re-targetability

Recent activity

- Follow us on http://www.github.com/MetaScale/nt2
- Prototype for single source GPU support
- Toward a global generic approach to parallelism
Thanks for your attention