(Costless) Software Abstractions for Parallel Architectures

numscale
Unlocked software performance

Joel Falcou

NumScale SAS – LRI - INRIA

CppCon – 01/07/2014

Context

Decades of hardware improvements

- Scientific Computing now drives most hardware innovations
- Current Solution: Parallel architectures
- Machines become more and more complex

Context

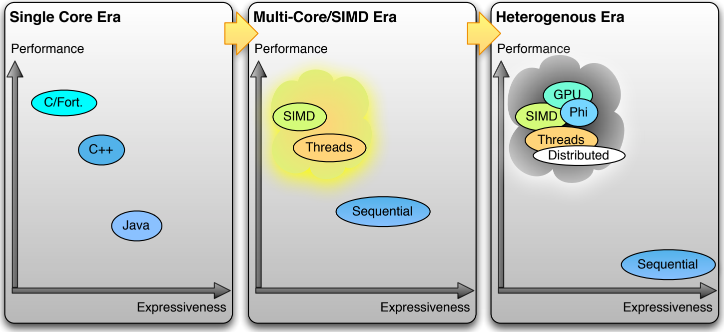
Decades of hardware improvements

- Scientific Computing now drives most hardware innovations
- Current Solution: Parallel architectures
- Machines become more and more complex

Example : A simple laptop

- CPU: Intel Core i5-2410M (2.3 GHz) : 4 logical cores, AVX
  - 4 logical cores
  - SIMD Extensions: SSE2-SSE4.2, AVX
- GPU: NVIDIA GeForce GT 520M (48 CUDA cores)

The Real Challenge of HPC



Designing tools for Scientific Computing

Challenges

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape
5. Be efficient

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape
5. Be efficient

Our Approach

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape
5. Be efficient

Our Approach

- Design tools as C++ libraries (1)

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape
5. Be efficient

Our Approach

- Design tools as C++ libraries (1)
- Design these libraries as Domain Specific Embedded Language (DSEL) (2+3)

Designing tools for Scientific Computing

Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape
5. Be efficient

Our Approach

- Design tools as C++ libraries (1)
- Design these libraries as Domain Specific Embedded Language (DSEL) (2+3)
- Use Parallel Skeletons as parallel components (4)

Designing tools for Scientific Computing

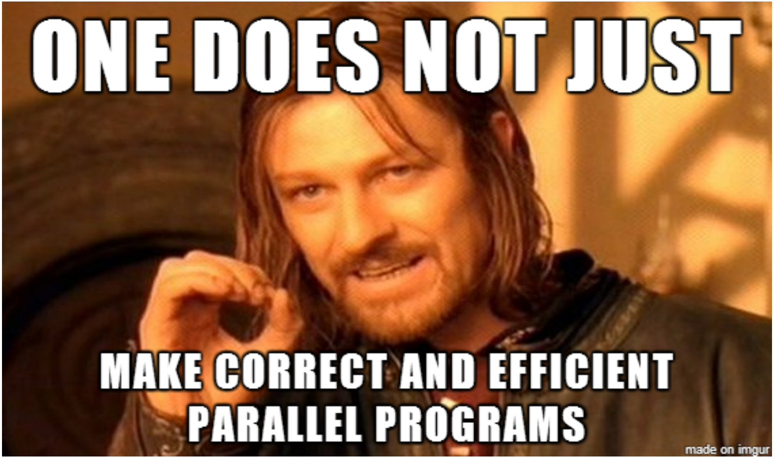
Challenges

1. Be non-disruptive
2. Domain driven optimizations
3. Provide intuitive API for the user
4. Support a wide architectural landscape
5. Be efficient

Our Approach

- Design tools as C++ libraries (1)
- Design these libraries as Domain Specific Embedded Language (DSEL) (2+3)
- Use Parallel Skeletons as parallel components (4)
- Use Generative Programming to deliver performance (5)

Parallel Programming Ain't Easy



Spotting abstraction when you see one

Why Parallel Programming Models ?

- Unstructured parallelism is error-prone
- Low level parallel tools are non-composable

Spotting abstraction when you see one

Why Parallel Programming Models ?

- Unstructured parallelism is error-prone
- Low level parallel tools are non-composable

Available Models

- Performance centric: P-RAM, LOG-P, BSP
- Data centric: HTA, PGAS
- Pattern centric: Actors, Skeletons

Spotting abstraction when you see one

Why Parallel Programming Models ?

- Unstructured parallelism is error-prone
- Low level parallel tools are non-composable

Available Models

- Performance centric: P-RAM, LOG-P, BSP
- Data centric: HTA, PGAS
- Pattern centric: Actors, Skeletons

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

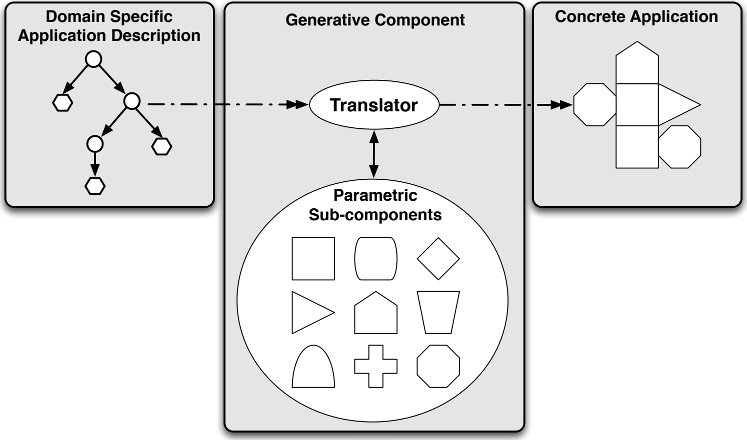
Software Abstraction

- 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

Generative 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

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.

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.

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

Embedded Domain Specific Languages

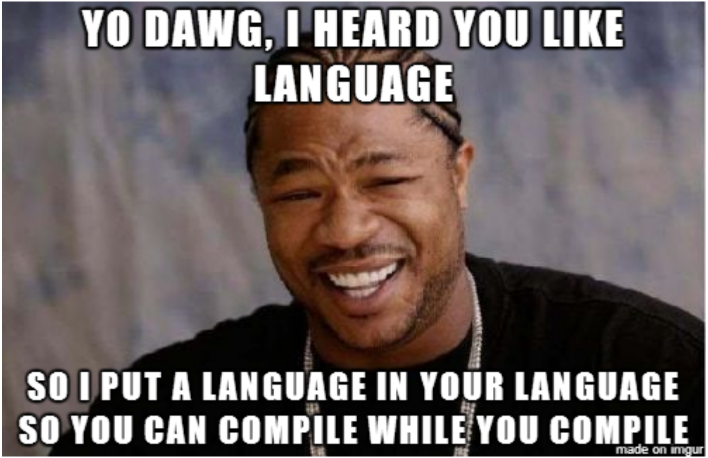
EDSL in C++

- Relies on operator overload abuse – see BOOST.PROTO
- 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

Expression Templates in A Nutshell

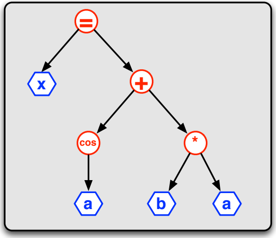


Expression Templates

```
matrix x(h,w),a(h,w),b(h,w);

x = cos(a) + (b*a);
```

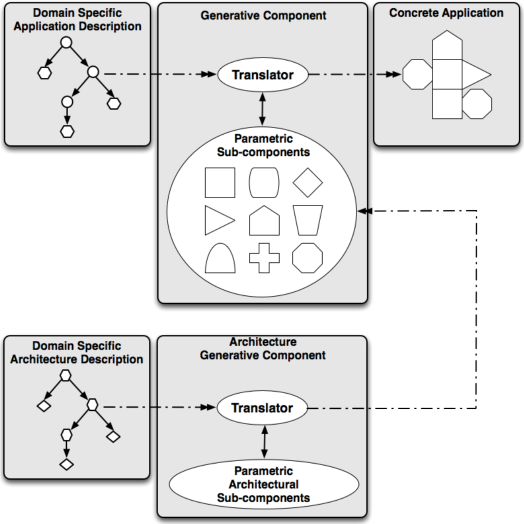
```
expr<assign
    ,expr<matrix&>
    ,expr<plus
          , expr<cos
                ,expr<matrix&>
                >
          , expr<multiplies
                ,expr<matrix&>
                ,expr<matrix&>
                >
          >(x,a,b);
```

Arbitrary Transforms applied on the meta-AST




```
#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)
           );
  }
}
```

Architecture Aware Generative Programming



Parallel DSEL in practice

Objectives

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

Parallel DSEL in practice

Objectives

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

Our contribution

- BSP++ : Generic C++ BSP for shared/distributed memory
- Quaff: DSEL for skeleton programming
- Boost.SIMD: DSEL for portable SIMD programming
- NT2: Matlab like DSEL for scientific computing

Parallel DSEL in practice

Objectives

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

Our contribution

- BSP++ : Generic C++ BSP for shared/distributed memory
- Quaff: DSEL for skeleton programming
- Boost.SIMD: DSEL for portable SIMD programming
- NT2: Matlab like DSEL for scientific computing

NT^2

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
- Code is made independant of architecture and runtime

The Numerical Template Toolbox

Comparison to other libraries

Feature	Armadillo	Blaze	Eigen	MTL	uBlas	NT^2
MATLAB-like API	✓	—	—	—	—	✓
BLAS/LAPACK binding	✓	✓	✓	✓	✓	✓
MAGMA binding	—	—	—	—	—	✓
SSE2+ support	✓	✓	✓	—	—	✓
AVX support	✓	✓	—	—	—	✓
AVX2 support	—	—	—	—	—	✓
Xeon Phi support	—	—	—	—	—	✓
Altivec support	—	—	✓	—	—	✓
ARM support	—	—	✓	—	—	✓
Threading support	—	—	—	—	—	✓
CUDA support	—	—	—	—	—	✓

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

- `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 works

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

- 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 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 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 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!

NT2 - From MATLAB ...

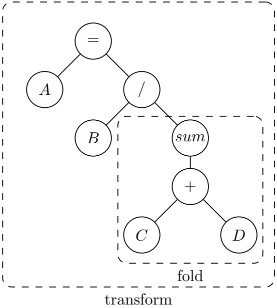
```
A1  = 1:1000;
A2  = A1 + randn(size(A1));

X = lu(A1*A1');

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

NT2 - ... to C++

```
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) );
```

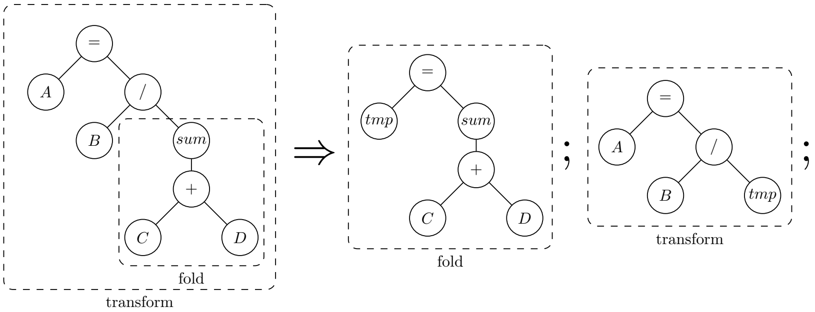
Parallel Skeletons extraction process

```
A = B / sum(C+D);
```



Parallel Skeletons extraction process

```
A = B / sum(C+D);
```



From data to task parallelism

Limits of the fork-join model

- Synchronization cost due to implicit barriers
- Under-exploitation of potential parallelism
- Poor data locality and no inter-statement optimization

From data to task parallelism

Limits of the fork-join model

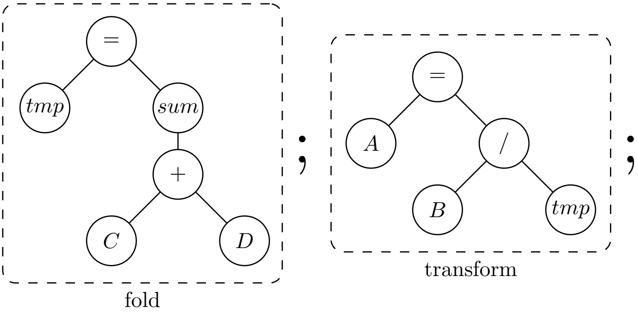
- Synchronization cost due to implicit barriers
- Under-exploitation of potential parallelism
- Poor data locality and no inter-statement optimization

From Skeletons to Actors

- Upgrade NT^2 to enable task parallelism
- Adapt current skeletons for taskification
- Use Futures (STD or HPX)to automatically create pipelines
- Derive a dependency graph between statements

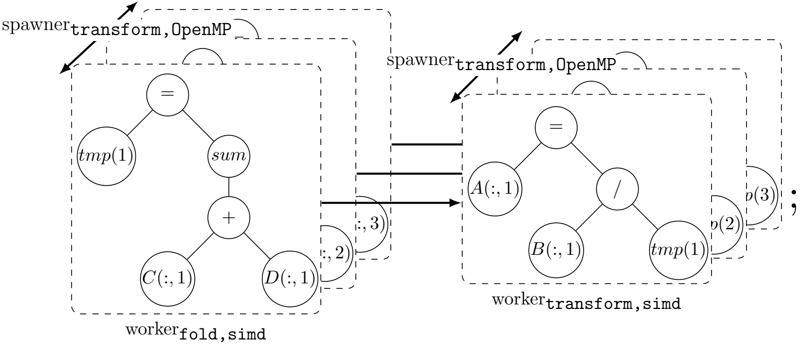
Parallel Skeletons extraction process - Take 2

```
A = B / sum(C+D);
```



Parallel Skeletons extraction process - Take 2

```
A = B / sum(C+D);
```



Sigma-Delta Motion Detection

Context

- Mono-modal algorithm based on background substraction
- Use local gaussian model of lightness variation to detect motion
- Target applications: robotic, video survey and analytics, defence
- **Challenge:** Very low arithmetic density
- **Challenge:** Integer-based implementation with small range

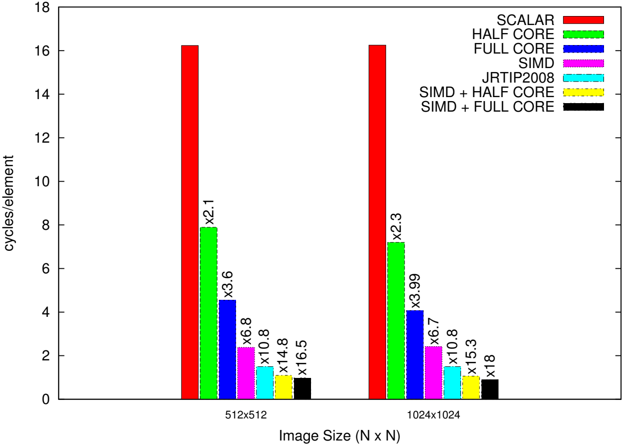
Motion Detection

NT^2 Code

```
table<char> sigma_delta( table<char>& background
                       , table<char> const& frame
                       , table<char>& variance
                       )
{
  // Estimate Raw Movement
  background = selinc( background < frame
                 , seldec(background > frame, background)
                 );

  table<char> diff = dist(background, frame);

  // Compute Local Variance
  table<char> sig3 = muls(diff,3);

  var = if_else( diff != 0
               , selinc( variance < sig3
                       , seldec( var > sig3, variance)
                       )
               , variance
               );

  // Generate Movement Label
  return if_zero_else_one( diff < variance );
}
```

Motion Detection

Performance



Black and Scholes Option Pricing

NT^2 Code

```
table<float> blackscholes( table<float> const& Sa, table<float> const& Xa
                         , table<float> const& Ta
                         , table<float> const& ra, table<float> const& va
                         )
{
  table<float> da = sqrt(Ta);
  table<float> d1 = log(Sa/Xa) + (sqr(va)*0.5f+ra)*Ta/(va*da);
  table<float> d2 = d1-va*da;

  return Sa*normcdf(d1)- Xa*exp(-ra*Ta)*normcdf(d2);
}
```

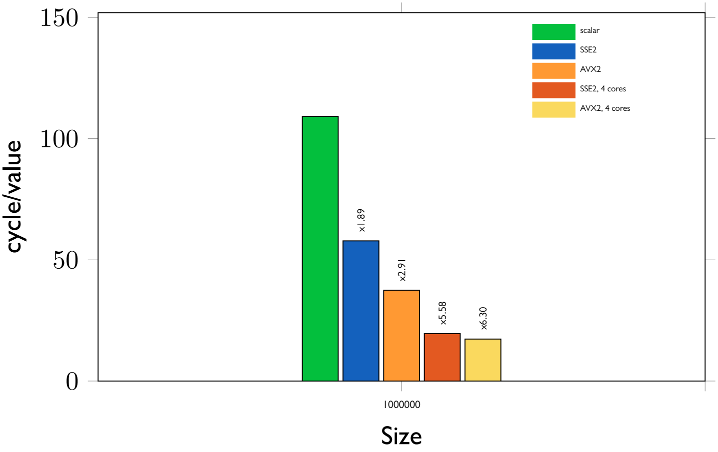
Black and Scholes Option Pricing

NT^2 Code with loop fusion

```
table<float> blackscholes( table<float> const& Sa, table<float> const& Xa
                         , table<float> const& Ta
                         , table<float> const& ra, table<float> const& va
                         )
{
  // Preallocate temporary tables
  table<float> da(extent(Ta)), d1(extent(Ta)), d2(extent(Ta)), R(extent(Ta));

  // tie merge loop nest and increase cache locality
  tie(da,d1,d2,R) = tie( sqrt(Ta)
                       , log(Sa/Xa) + (sqr(va)*0.5f+ra)*Ta/(va*da)
                       , d1-va*da
                       , Sa*normcdf(d1)- Xa*exp(-ra*Ta)*normcdf(d2)
                       );

  return R;
}
```

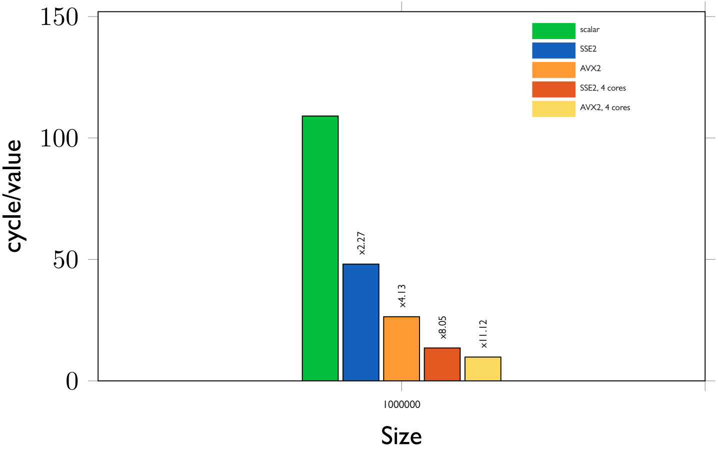
Black and Scholes Option Pricing

Performance



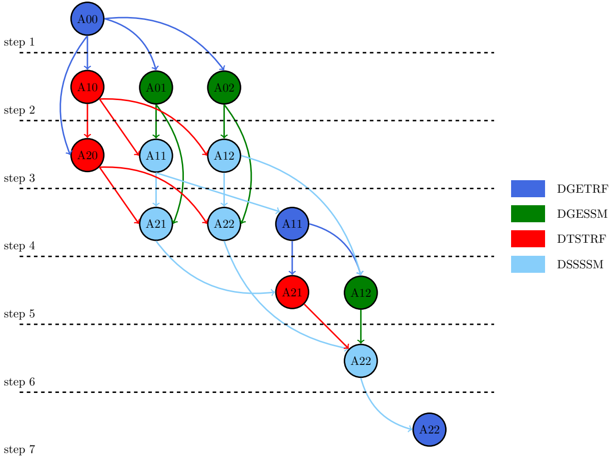
Black and Scholes Option Pricing

Performance with loop fusion



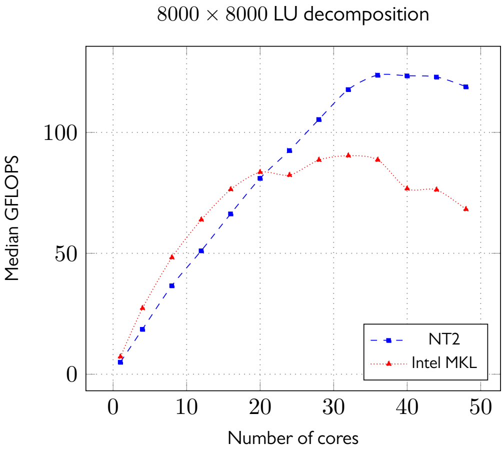
LU Decomposition

Algorithm



LU Decomposition

Performance



What we learn

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

What we learn

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

New Directions

- Toward a global generic approach to parallelism
- Turning hacks into language features

Generic Parallelism

Parallel C++ Concepts

- Expand function hierarchization to Concepts
- e.g : DataParallel, AssociativeOperations, etc.
- Use C++1y Concept overloading to split skeletons

Generic Parallelism

Parallel C++ Concepts

- Expand function hierarchization to Concepts
- e.g : DataParallel, AssociativeOperations, etc.
- Use C++1y Concept overloading to split skeletons

Impact

- Less work for the Skeleton users
- Extendable through refinement
- Static assertion of function properties

New C++ Language Features

My C++ Christmas Land

- Build lazy evaluation into the language
- Interactions with generic function is cumbersome
- SIMD as part of the standard at type level

New C++ Language Features

My C++ Christmas Land

- Build lazy evaluation into the language
- Interactions with generic function is cumbersome
- SIMD as part of the standard at type level

Current Work

- Can `sizeof` inspires an `ast_of` operator
- Proposal N4035 for `auto` customization
- Proposal N3571 for standard SIMD computation

Perspectives

At tools level

- Prototype of single source GPU support
- Work on distributed systems
- Applications to Big Data

Perspectives

At tools level

- Prototype of single source GPU support
- Work on distributed systems
- Applications to Big Data

At language level

- Formalize meta-programming
- DSEL verification transferance over C++
- Interaction with polyhedral model

Thanks for your attention