Fast numerical computation in C++: Expression Templates and Beyond to Lazy Code Generation (LzCG)

B. Nikolic

Cavendish Laboratory/Kavli Institute
University of Cambridge

BoostCon 2011
May 2011

Overview of ideas

1. 'Standard' rules of C++ lead to inefficient numerical code
2. New rules ($\equiv$ sub-languages) can be implemented using expression templates
   - 2.1 Types are used confer information about expressions
   - 2.2 Translated to 'standard' C++ at compile-time
3. Makes high-performance numerical C++ libraries possible and successful
4. But is it enough?
   - 4.1 Most efficient algorithm not obvious at compile-time
   - 4.2 Convenience/flexibility of generating code in C++
5. Types *retain* information about expressions in signatures in object code
   - 5.1 Can re-generate expression template implementations post-compilation-time

Outline

About myself: ALMA telescope

Largest ground-based astronomy project in the world

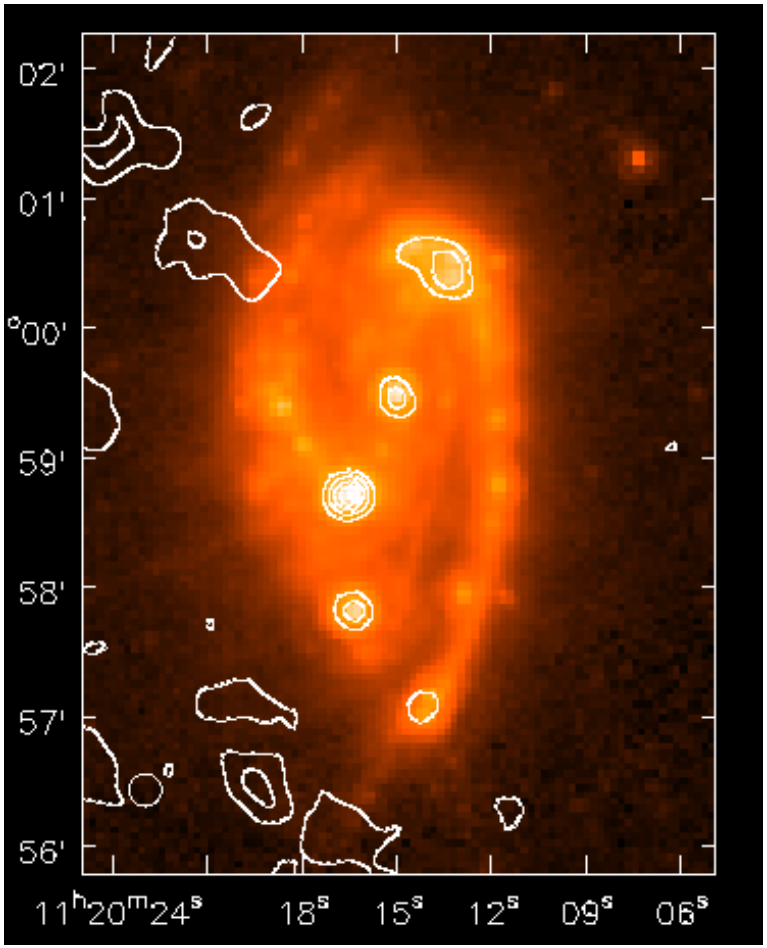
Currently being commissioned at altitude of 5000 m in Chile. Will have 66 telescopes separated by up to 15 kms and observed at wavelength between 7 and 0.35 mm.

About myself: Green Bank Telescope

Largest steerable telescope in the world

Main reflector is 100x110 m in size, total height 160 m.
Entire structure is accurate to 0.25 mm.

About myself: Thermal radio emission from Messier 66



- Colour scale is emission from dust at 0.024 mm wavelegnth
- Contours represent emission at 3 mm from hot electron gas
- Both appear to be powered by recent star formation

General Interests

- Model optimisation and statistical inference (maximum-likelihood, Markov Chain Monte Carlo, Nested Sampling techniques)
- Pricing and risk-management of derivative contracts
- Remote sensing of Earth's atmosphere
- Radiative transfer and other physical simulations

$\Rightarrow$ All very numerically intensive applications...

Aperture synthesis radio-astronomy

- Revolutionised the radio view of the universe – Nobel prize in 1972
- Development of the technique closely tied to computers:
  - Lots of Fourier Transforms
  - Large quantities of data to be binned, inspected, discarded if necessary
  - Instruments inherently unstable so calibration is critical
- Atacama Large Millimetre Array: eventually 66 antennas, $\sim 20\,\mathrm{Mb/s}$ average output data rate:
  - Computational issues inconvenient, reduce scientist productivity
- Square Kilometre Array (SKA): 1000s antennas, wide field of view, $\sim \mathrm{few}\,\mathrm{Gb/s}$ average output data rate:
  - Computational issues limiting factor in scientific output

Risk management of 'derivative' contracts in finance

Requirements in just one product line (e.g., credit derivatives)

Typically calculations involve either: solving PDEs using finite differences; or computing FFTs; or Monte-Carlo (MC) simulations.

- $2000\,\text{nodes} \times 1\,\text{kW/node} + 50\%\,\text{aircon cost} = 3\,\text{MW}$
- $3\,\text{MW} \times 10\,\text{p/s} \times 8500\,\text{hr/yr} = \sim 2.5 \times 10^6\,\text{GBP/yr}!$
- Additional costs $\propto$ number of nodes:
  - Installation, maintenance, software licenses (even Excel sometimes!)
  - Floor-space (in expensive buildings)
  - Standby backup power generation costs

Numerical performance

(Why) does it matter?

Easily parallelisable

- Cost
- Heat, power, floor space
- Environmental impact
- Time to scale-up
- Access to capital

Difficult to parallelise

- Feasibility
- Latency
- User patience

Parallelisation is usually **the** most important aspect of high-performance numerical computing

- Not directly considering it in this talk although much of the material is relevant

Small problems ≡ simple solutions

Many practical scientific and industrial problems can be accelerated a simple way

Listing 1: By-hand coding + SIMD intrinsics

```
void add2Vect(const std::vector<double> &v1,
              const std::vector<double> &v2,
              std::vector<double> &res) {
  typedef double v2df __attribute__ ((mode(V2DF)));
  v2df * dest=(v2df *)&(*res.begin());
  const size_t n=v1.size();
  const v2df *src1=(const v2df *)&v1[0];
  const v2df *src2=(const v2df *)&v2[0];
  if (n%2==0)
    {
    for(size_t i=0; i<n/2; i++)
    {
        dest[i]=__builtin_ia32_addpd(src1[i],src2[i]);
    }
  }
  else
  {
    for(size_t i=0; i<n; ++i)
    {
      dest[i]=src1[i]+src2[i];
    }
  }
}
```

Simple problems are common in real life but not really the subject of this talk!

Hand coding unsuitable for *large* systems

- Correctness
- Maintainability, readability, portability
- Algorithms need adjustment over time
- Experiment with different implementations of algorithms
- Approximations: how much precision, what accuracy is necessary?

$\Rightarrow$ These can be difficult to achieve with complex hand-crafted code!

Warning!

“Don’t try this at home” – try existing libraries first

Writing numerical libraries is difficult and error prone – always carefully consider alternatives!

- Can you use standard existing libraries (“C” or “C++”)
- Are you writing a general purpose library or an application?
- Can you, in advance, identify a subset of algorithm which is likely to consume most time but can present a clean, data-only, interface?

Outline

Requirements for good numerical performance

- Maximise parallelism
  - Use all of the nodes/processors/cores/execution units
  - Use Single-Instruction-Multiple-Data (SIMD)
- Minimise memory access
  - Keep close data to be processed together
  - Use algorithms that process small chunks of input data at a time
  - Avoid temporaries
- Minimise 'branching'
  - Keep the pipeline and speculative fetches good
  - But, need enough code at hand to execute
- Minimise quantity of transcendental calculations
  - Includes division in this set
  - Reducing precision or accuracy makes these faster

Optimisation Challenges I

- Want: to describe the algorithm in simple, readable, re-usable way

```
// This :
R=A+B+C+D+E;
// Not this :
addFiveVect_Double_Double_Double_Double_Double(A, B, C, D, E, R);
```

- Rules for transforming such description to executable code need to be complex to be efficient
- Simple application of rules applying to C++ **objects**:
  - Arguments are 'evaluated' before being passed to functions
  - Operators take two arguments at most
  - Creation of temporaries
  - Iteration is interpreted literally as ordered repetition of same segment of code

Fundamentally: One step of the algorithm at a time
Definitely not suitable for fast code!

Optimisation Challenges II

- Programs may be compiled on one hardware setup but run on many different hardware setups
- Might need (or want) to adjust rules for generation of implementations after the compilation of the main program
- Speed of execution of particular implementation of algorithm can be difficult to predict
  - Depends on precise model of the processor: clock speed, number of floating point execution cores, hyper-threading, branch-prediction, pipeline designs, microcode implementations of complex instructions
  - Sizes of the various levels of data *and* code caches, main memory bus speed

Example: row vs column matrix access

Listing 2: Sum by iterating through columns first

```
u::matrix<double> A(nrow, ncol);
// initialise...
double res;
for(size_t k=0; k<repeat; ++k)
  for(size_t i=0; i<nrow; ++i)
    for(size_t j=0; j<ncol; ++j)
      res+=A(i,j);
return res;
```

Example: row vs column matrix access

Listing 3: Sum by iterating through rows first

```
u::matrix<double> A(nrow, ncol);
// initialise...
double res;
for(size_t k=0; k<repeat; ++k)
  for(size_t j=0; j<ncol; ++j)   // note swap
    for(size_t i=0; i<nrow; ++i)// note swap
      res+=A(i,j);
return res;
```

Example: row vs column matrix access

Rows	Columns	Time for col-first (seconds)	Time for row-first (seconds)
1000000	10	4.16	4.52
100000	100	4.15	9.54
10000	1000	4.04	5.52
1000	10000	4.02	5.41
100	100000	4.00	4.56
10	1000000	3.96	4.04

Note

- Compiled using gcc without optimisation
- Run on my laptop
- $\Rightarrow$ Illustration only!

Techniques used for advanced numerical libraries

- Optimising compilers
- Custom compilers for standard languages
- Code generation using custom languages/frameworks
- Run-time selection according to detected hardware
- Run-time profiling of multiple/many algorithms
- Run-time generation of machine code

Expression templates and lazy code generation can adapt all of these to standard C++.

A quick case study: FFTW

http://www.fftw.org

Revolutionary at the time:

- Building blocks ("codelets") of algorithms generated at compile-time:
  OCAML $\rightarrow$ C $\rightarrow$ machine code
- Run-time selection of best combination of building-blocks
  - Memory-layout, cache sizes, relative speed of memory
  - Code selection can be saved
- SIMD+ (p)threads + MPI parallelism
- Presents trivial C-language & Fortran-language interfaces

Some subtleties

- The order of floating point operations usually matters even when operations on real numbers would not:

$$1 + (-1 + 10^{-10}) \neq (1 - 1) + 10^{-10} \quad (1)$$

- "Standard" x86 floating point uses 80-bit internal precision! (SIMD instructions do not)
- Non-normal (NaN, Inf, de-normalised) floating point number badly affect performance:

$$\infty + 1 = \infty \quad \text{(slowly!)} \quad (2)$$

- Transcendentals can be approximated to less than full precision

$\Rightarrow$ Ensuring "bit-equivalent" results is difficult and expensive

Outline

How to identify bottlenecks

- Run a tick counting profiler: `http://oprofile.sourceforge.net/`
  - Get a stochastic measurement of where the CPU spends most of its time without modifying the code!
- Run a call-graph profiler: `http://valgrind.org/`
  - Shows how the CPU-intensive parts of the code fit into the big picture of the application
- Compile to assembly only (“gcc -S”) and look at the code!
  - Allows identifications of in-efficiencies in the produced code and gives hints for optimisations

Outline

What is an expression template?

Expression template =

- Templated
- Type
- Where the **type** itself encodes an **operation**, **expression** or **algorithm**
- Passed between functions as instantiated objects (usually with data as references)
- Implemented through (partial) specialisations

Expression template (minimal) example

Listing 4: Adding vectors

```
// Some vectors to use in the example
std::vector<double> a(10, 1.0), b(10, 2.0);

// Addition the old way (see next slide)
double res=(a+b+a+b)[3];
```

1. A temporary is created
2. All members of the result vector are computed
3. The temporary is iterated three times over

Helper function

Listing 5: Simple add operator for `std::vector`

```
template< class T>
std::vector<T>
operator+(const std::vector<T> &a,
          const std::vector<T> &b)
{
  std::vector<T> res(a.size());
  for(size_t i=0; i<a.size(); ++i)
    res[i]=a[i]+b[i];
  return res;
}
```

1. Considers two vectors at a time
2. Iterates through the whole vector and computes the entire result

Expression template (minimal) example

Listing 6: Expression template class

```
template<class E1, class E2,
         class op=valueop>
struct binop
{
  const E1 &left;  const E2 &right;

  binop(const E1 &left, const E2 &right,
        op opval):
    left(left), right(right) {};
  binop(const E1 &val);
};
```

1. The sub-expression are referenced in **left**, **right**
2. The operation is contained in type of **op**
   - **valueop, addop** are simple tag structs

Expression template (minimal) example

Listing 7: Creation of compound expression

```
template<class E1, class E2>
binop<E1, E2, addop>
operator+ (const E1 &left,
           const E2 &right)
{
  return binop<E1, E2, addop>(left,
                              right,
                              addop());
}
```

1. E1, E2 are ‘free’ template parameters – types of sub-expression is encoded in result
2. `addop` is the type of third template parameter – encodes addition of sub-expressions

Expression template (minimal) example

Listing 8: Templated evaluation operation

```
/// Evaluate the i-th element of
/// an expression
template<class E1, class E2, class op>
double eval(const binop<E1, E2, op> &o,
            size_t i);
```

Expression template (minimal) example

Partial specialisation for add operation

Listing 9: Implementation of sum operation using partial specialisation

```
template<class E1, class E2>
double eval(const binop<E1, E2, addop> &o,
            size_t i)
{
  return eval(o.left, i)+eval(o.right, i);
};
```

1. Specialised on **addop** as third temp-par
2. Recursively **eval**uate and add using standard **operator+**

Expression template (minimal) example

Partial specialisation for value operation

Listing 10: Partial specialisation for value types

```
template<class T>
double eval(const binop<T, T,  valueop> &o,
            size_t i)
{
  return o.left[i];
};
```

1. Specialised on **valueop** as third template-parameter
2. Simply returns the value of reference vector at **i**

Expression template (minimal) example

Listing 11: In use

```
std::vector<double> a(10, 1.0),
                    b(10, 2.0);
binop<> ba(a), bb(b);

double res=eval(ba+bb+ba+bb, 3);
```

1. Does not create a temporary vector
2. Evaluates only the third element of the result

A look at the types

Listing 12: Signatures as seen by `nm -C`

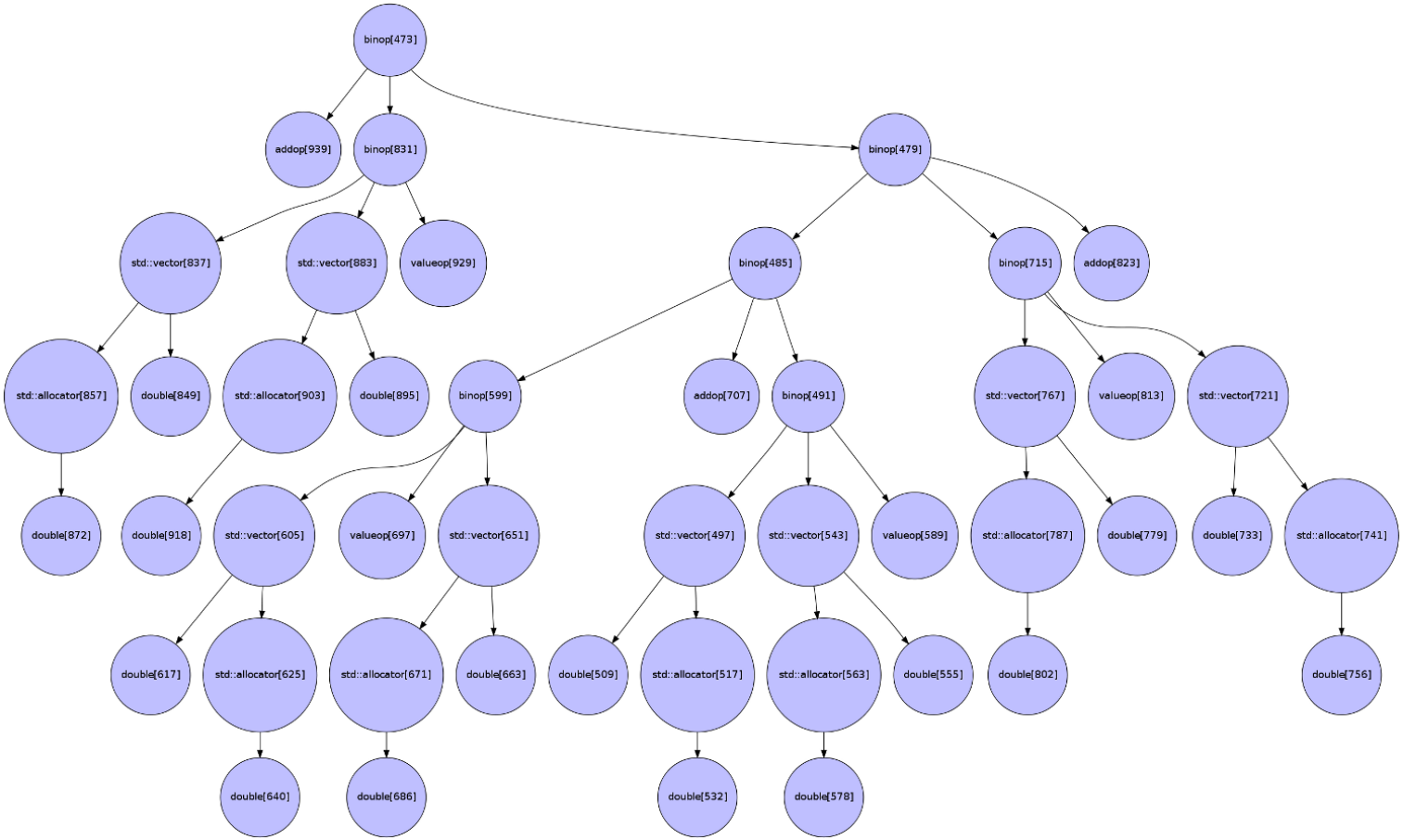
```
000000000040114a W double eval<binop<binop<binop<std::vector<double, std::allocator<double> >, st
00000000004014a7 W double eval<binop<binop<std::vector<double, std::allocator<double> >, std::vec
0000000000401624 W double eval<binop<std::vector<double, std::allocator<double> >, std::vector<do
0000000000401473 W double eval<std::vector<double,std::allocator<double> >, std::vector<double, s
```

A look at the types

Listing 13: Output of `nm -C` wrapped properly

```
double eval<binop<binop<binop<std::vector<double, std::allocator<double> >,
                              std::vector<double, std::allocator<double> >,
                              valueop>,
                        binop<std::vector<double, std::allocator<double> >,
                              std::vector<double, std::allocator<double> >,
                              valueop>,
                        addop>,
                  binop<std::vector<double, std::allocator<double> >,
                        std::vector<double, std::allocator<double> >,
                        valueop>,
                  addop>,
            binop<std::vector<double, std::allocator<double> >,
                  std::vector<double, std::allocator<double> >,
                  valueop> >(
binop<binop<binop<binop<std::vector<double, std::allocator<double> >,
                              std::vector<double, std::allocator<double> >,
                              valueop>,
                        binop<std::vector<double, std::allocator<double> >,
                              std::vector<double, std::allocator<double> >,
                              valueop>,
                        addop>,
                  binop<std::vector<double, std::allocator<double> >,
                        std::vector<double, std::allocator<double> >,
                        valueop>,
                  addop>,
            binop<std::vector<double, std::allocator<double> >,
                  std::vector<double, std::allocator<double> >,
                  valueop>,
            addop> const&,
unsigned long)
```

A look at the types



Lazy evaluation

Not the same as lazy code generation...

In summary:

1. Operations (+function calls) return ‘expressions’ not results
2. The order *and* implementation of operations in expression can be modified (at compile-time)
3. Results are only evaluated at a *boundary*, e.g., when assigning to plain-old-data
4. If the result is never required, it is never computed

Doing this properly is very elegant but things get complicated – see the Haskel programming language

Things to keep in mind:

1. Side-effects are ill-defined – stick to ‘functional’ programming
   - Assigning to an already initialised variable is not functional programming!
2. Implemented with expression templates, size of types grows very quickly

Libraries

- Eigen `http://eigen.tuxfamily.org` (LGPL)
  Array Ops, Basic + Advanced Linear Algebra, Geometry
- Armadillo `http://arma.sourceforge.net/` (LGPL)
  Array Ops, Basic + Advanced Linear Algebra, Geometry
- Boost.uBLAS `http://www.boost.org` (Boost License)
  Basic Linear Algebra
- NT^2 `http://nt2.sourceforge.net/` (LGPL)
- Blitz++ `http://www.oonumerics.org/blitz/` (GPL+artistic)
  This was one of the first libraries to use expression templates

Outline

Introduction

‘Standard’ expression templates

1. Types convey information about algorithm that functions implement
2. These types are interpreted at *compile-time* and corresponding code generated

Lazy code generation

1. The types are recorded in object code too, so the algorithm implemented by symbols is retained
2. Generate new implementations, *post-compilation-time*

$\Rightarrow$ Introduces new flexibility and modularity in code generation process

Example – BLAS Reference card

Level 1 BLAS

dim scalar vector vector scalars 5-element array		prefixes
SUBROUTINE xROTG (A, B, C, S)	Generate plane rotation	S, D
SUBROUTINE xROTMG(D1, D2, A, B, PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT (N, X, INCX, Y, INCY, C, S)	Apply plane rotation	S, D
SUBROUTINE xROTM (N, X, INCX, Y, INCY, PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP (N, X, INCX, Y, INCY)	$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL (N, ALPHA, X, INCX)	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
SUBROUTINE xCOPY (N, X, INCX, Y, INCY)	$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY (N, ALPHA, X, INCX, Y, INCY)	$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT (N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU (N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC (N, X, INCX, Y, INCY)	$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT (N, X, INCX, Y, INCY)	$dot \leftarrow \alpha + x^T y$	SDS
FUNCTION xNRM2 (N, X, INCX)	$nrm2 \leftarrow \|x\|_2$	S, D, SC, DZ
FUNCTION xASUM (N, X, INCX)	$asum \leftarrow \|re(x)\|_1 + \|im(x)\|_1$	S, D, SC, DZ
FUNCTION IxAMAX(N, X, INCX)	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

options dim b-width scalar matrix vector scalar vector		
xGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xGBMV (TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xHEMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xSYMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xTRMV (UPLO, TRANS, DIAG, N, A, LDA, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTBMV (UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTPMV (UPLO, TRANS, DIAG, N, AP, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTRSV (UPLO, TRANS, DIAG, N, A, LDA, X, INCX)	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTBSV (UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX)	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTPSV (UPLO, TRANS, DIAG, N, AP, X, INCX)	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
options dim scalar vector vector matrix		
xGER (M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
xGERU (M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
xGERC (M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
xHER (UPLO, N, ALPHA, X, INCX, A, LDA)	$A \leftarrow \alpha xx^H + A$	C, Z
xHPR (UPLO, N, ALPHA, X, INCX, AP)	$A \leftarrow \alpha xx^H + A$	C, Z
xHER2 (UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xHPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, AP)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xSYR (UPLO, N, ALPHA, X, INCX, A, LDA)	$A \leftarrow \alpha xx^T + A$	S, D
xSPR (UPLO, N, ALPHA, X, INCX, AP)	$A \leftarrow \alpha xx^T + A$	S, D
xSYR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
xSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, AP)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Level 3 BLAS

options dim scalar matrix matrix scalar matrix		
xGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
xSYMM (SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
xHEMM (SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
xSYRK (UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
xHERK (UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
xSYR2K(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$	S, D, C, Z
xHER2K(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$	C, Z
xTRMM (SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
xTRSM (SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

BLAS – structure of function of names

Operation:

DOT	scalar product
AXPY	vector sum
MV	matrix-vector product
SV	matrix-vector solve
MM	matrix-matrix product
SM	matrix-matrix solve

Numerical type:

S	single real
D	double real
C	single complex
Z	double complex

Matrix type:

GE	general
GB	general band
SY	symmetric
SB	symmetric band
SP	symmetric packed
HE	hermitian
HB	hermitian band
HP	hermitian packed
TR	triangular
TB	triangular band
TP	triangular packed

C++ type is encoded in the function (*symbol*) name

Listing 14: Back to basic expression template example

```
double eval<binop<binop<binop<std::vector<double, std::allocator<double> >,
                              std::vector<double, std::allocator<double> >,
                              valueop>,
                        binop<std::vector<double, std::allocator<double> >,
                              std::vector<double, std::allocator<double> >,
                              valueop>,
                        addop>,
                  binop<std::vector<double, std::allocator<double> >,
                        std::vector<double, std::allocator<double> >,
                        valueop>,
                  addop>,
            binop<std::vector<double, std::allocator<double> >,
                  std::vector<double, std::allocator<double> >,
                  valueop> >(
binop<binop<binop<binop<std::vector<double, std::allocator<double> >,
                        std::vector<double, std::allocator<double> >,
                        valueop>,
                  binop<std::vector<double, std::allocator<double> >,
                        std::vector<double, std::allocator<double> >,
                        valueop>,
                  addop>,
            binop<std::vector<double, std::allocator<double> >,
                  std::vector<double, std::allocator<double> >,
                  valueop>,
            addop>,
      binop<std::vector<double, std::allocator<double> >,
            std::vector<double, std::allocator<double> >,
            valueop>,
      addop> const&,
```

C++ type is encoded in the function (*symbol*) name

Listing 15: Back to basic expression template example (raw `nm`)

```
000000000040114a W _Z4evalI5binopIS0_IS0_ISt6vectorIdSaIdEES3_\
                   7valueopES5_5addopES5_S6_ES5_EdRKS0_IT_T0_S6_Em
```

- The function/symbol name specifies *exactly* the algorithm that it should apply to its data
- This information is available *post-compile-time* (as simply as using `nm`)
- Implementation can be generated post-compilation and used in program simply by linking it (*weak* symbols make this trivial)

Alternative view

What we get:

Expression templates allow export of a subset of the program parse tree outside the compiler environment.

What would be the alternative?

Parsing the C++ source code $\equiv$ writing new compiler

Potential advantages of LzCG

General:

1. A very simple, clean, efficient mechanism for separating specification of *what* needs to be from *how* its done
2. A mechanism introducing an embedded language in C++ that can be implemented outside traditional C++ compilation scheme

Specific:

1. Can try multiple algorithms and select the experimentally most efficient
2. Can detect hardware configuration and generate efficient code without access to source code
3. Can use custom and third-party code generators (GPU compilers, cluster compute tools, or Ocaml + C-compiler!)

Outline

Example introduction – Fast Fourier Transforms

- Optimum FFT algorithm depends on *array length*, cache sizes, processor architecture, etc., etc
- Normally selected by benchmarking at run-time
- Can we do better if we know array size at compile-time?

Code

Listing 16: Call to compute the forward FFT

```
boost::array<double, 10> inp;
boost::array<double, 10> out;

FFTForward(inp, out);
```

1. Array sizes are known at compile-time
2. Can select optimum algorithm as soon as we know what machine we run on
   - This may be after compile-time but *must* be before run-time
3. Selection before run-time:
   - 3.1 Removes potentially lengthy run-time algorithm selection
   - 3.2 Makes the program performance more predictable
   - 3.3 Reduces code size
   - 3.4 Allow selection from wider range of algorithms

Code

Listing 17: Call to FFTW the old-fashioned way

```
fftw_plan  p=
fftw_plan_dft_1d(5,
                 (fftw_complex*)(&in[0]),
                 (fftw_complex*)(&out[0]),
                 FFTW_FORWARD,
                 FFTW_ESTIMATE);
fftw_execute(p);
```

1. First call to create the plan can be time consuming (e.g., $\sim$ 1 second !)
2. Linking the entire library – 'codelets' + the algorithm selection code

Code

Listing 18: Declaration of the FFTForward template

```
template <class T, std::size_t N>
void FFTForward(const boost::array<T, N> &in,
                boost::array<T, N> &out);
```

Listing 19: Call to compute the forward FFT

```
boost::array<double, 10> inp;
boost::array<double, 10> out;

FFTForward(inp, out);
```

Code

Listing 20: Signature of the call to FFTForward (`nm -C`)

```
00000000004005e3 W void
FFTForward<double, 10ul>(boost::array<double, 10ul> const&,
                         boost::array<double, 10ul>&)
```

This is all the information we need to select an algorithm:

1. Parse signatures to identify all instances of FFTForward
2. Machine generate new C++ code with specialisation for *each* FFTForward instance but with optimum algorithm pre-selected
3. Compile and link these with original code

Note:

- Do not need access to the application source code – just the object file
- Can do the code generation on a different computer, using different tools, compilers & languages

Implementation

Listing 21: Parse symbol table

```
def analyseCal(fnamein):
    mre=re.compile(".*void␣FFTForward<double,␣(?P<N>\d+)ul>\\(\
boost::array<double,␣\d+ul>␣const&,␣\
boost::array<double,␣\d+ul>&\\).*")
    stable=subprocess.Popen(["nm", "-C", fnamein],
                            stdin=subprocess.PIPE,
                            stdout=subprocess.PIPE).communicate()[0]
    for l in stable.split("\n"):
        m=mre.match(l)
        if m:
            s=mkFunc(int(m.group("N")))
```

1. Simple regular expression on output of the `nm -C`
2. The array length is extracted from the signature

Implementation

Listing 22: Algorithm selection

```
    def templateProg(N):
    return """
#include <fftw3.h>
int main(void)
{
  const size_t N=%i;
  fftw_complex *in, *out;
  fftw_plan p;
  in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
  out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
  p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_MEASURE);
  fftw_print_plan(p); // This outputs the plan to stdout
}
""" % (N/2)
```

1. Trivial C program that calls FFTW with right array size
2. Execute this at lazy-code-generation-time
3. Print the selected best algorithm
4. Hardcode this algorithm selection in the specialised function

Implementation

Listing 23: Emit a specialised function for this array size

```
def mkFunc(N):
    mkProg(templateProg(N))
    plan=open("tmpProgOut").read()
    s=""
#include <boost/array.hpp>
#include "fftbind.hpp"
#include "fftw3.h"
template <>
void FFTForward<double, %i>(const boost::array<double, %i> &in,
boost::array<double, %i> &out)
{
const char *mplan="%s";
fftw_import_wisdom_from_string(mplan);
fftw_plan p=fftw_plan_dft_1d(%i,
(fftw_complex *)const_cast<double *>(&in[0]),
                             (fftw_complex *)(&out[0]),
                             FFTW_FORWARD,
                             FFTW_ESTIMATE);
fftw_execute(p);
fftw_destroy_plan(p);
}
""" % (N, N, N, plan, N/2)
    return s
```

1. The plan is stored as string literal within each specialised function

What have we achieved?

1. Optimum, pre-selected FFTW transform for each array of known size at compile-time – efficiency
2. Could switch to multi-core/GPU/etc *without* access to source code – modularity

Implementation shortcomings (this is a quick example!):

- The entire FFTW is linked-in – not just the specific algorithms
- Loading plans at run-time could be significant for small transforms

Outline

Simple C++ code is not numerically efficient

1. Function arguments evaluated promptly:

```
vector<double> myVect(10000000, 3.0);
takeOne(1.0/myVect, 5);
```

2. Only binary operators

```
vector<double> myVect(10000000, 3.0);
vector<double> myVect2=myVect+myVect+myVect;
```

3. Optimum algorithm sometimes can not be selected at compile-time

Expression templates resolve many of these issues

1. Expressions create objects with *type* that specifies the algorithm to be carried out
2. This *type* is interpreted at compile-time (through partial specialisation) to generate an efficient algorithm implementation of the whole expression

$\Rightarrow$ Blitz++, Boost::UBLAS, Armadillo, Eigen, NT^2

Lazy code generation

1. The *type* of expression templates is available in object code

```
00000000004005e3 W void
FFTForward<double, 10ul>(boost::array<double, 10ul> const&,
                         boost::array<double, 10ul>&,
                         boost::array<double, 10ul>&)
```

2. It can be interpreted post-compilation-time to generate code for the implementation
   - Try different algorithms
   - Delay algorithm selection for final hardware
   - Increased modularisation
3. The original implementation can be empty or a fall-back

When to use Lazy code generation

1. Need to do empirical selection of optimum algorithms
2. Want to do code generation using non-C++ compiler tools:
   - OcamL $\rightarrow$ C+intrinsics $\rightarrow$ machine code
   - GPU compiler?
3. Need to re-establish clean separation between application and library
   - Update implementations without access to original source code
   - Licensing concerns, proprietary libraries

Thanks for listening!

- See `http://www.bnikolic.co.uk/` and `http://www.mrao.cam.ac.uk/~bn204/` and for more information