Armadillo: Template Metaprogramming for Linear Algebra

Leo Antunes and Markus Kruber

Seminar on Automation, Compilers, and Code-Generation
HPAC

June 2014

What is Armadillo?

- C++ Library for Linear Algebra

What is Armadillo?

- C++ Library for Linear Algebra
- Aims for ease of use
- Still wants performance

What is Armadillo?

- C++ Library for Linear Algebra
- Aims for ease of use
- Still wants performance
- Abstract interface to BLAS and LAPACK

What is Armadillo?

- ▷ C++ Library for Linear Algebra
- ▷ Aims for ease of use
- ▷ Still wants performance
- ▷ Abstract interface to BLAS and LAPACK
- ▷ Pure template library (only headers)

Code example

Calculating $A^{-1}Bvv^T$:

```
mat     A = << 3.0 << 2.5 << endr
            << 2.0 << 1.0 << endr
mat     B = randu(2,5);
colvec  v = randu(5);

mat result = inv(A)*B*v*trans(v);

cout << "Result:" << endl << result.row(0) << endl;
```

Refresher: C++ Templates

- Allow type-agnostic development (e.g.: `class Array<type>`)
  "generics" in Java, C#; "parametric polymorphism" in Haskell, Scala

Refresher: C++ Templates

- Allow type-agnostic development (e.g.: `class Array<type>`)
  "generics" in Java, C#; "parametric polymorphism" in Haskell, Scala
- Realized at compile-time → **code generation**

Refresher: C++ Templates

- Allow type-agnostic development (e.g.: `class Array<type>`)
  "generics" in Java, C#; "parametric polymorphism" in Haskell, Scala
- Realized at compile-time → **code generation**
- Generate different class for each `type` used in code
  (thus: "template")

Refresher: C++ Templates

- Allow type-agnostic development (e.g.: `class Array<type>`)
  "generics" in Java, C#; "parametric polymorphism" in Haskell, Scala
- Realized at compile-time → **code generation**
- Generate different class for each `type` used in code
  (thus: "template")
- Therefore: no dynamic typing

Refresher: C++ Templates

- Allow type-agnostic development (e.g.: `class Array<type>`)
  "generics" in Java, C#; "parametric polymorphism" in Haskell, Scala
- Realized at compile-time → **code generation**
- Generate different class for each `type` used in code (thus: "template")
- Therefore: no dynamic typing
- Allows for further optimizations by compiler

C++ Template example

Type-dependent:

```
class Adder{
    int base;
    public:
    Adder(int x):base(x){}

    int doit(int y){
        return base+y;
    }
};
```

Type-agnostic:

```
template<typename TYPE>
class Adder{
    TYPE base;
    public:
    Adder(TYPE x):base(x){}

    TYPE doit(TYPE y){
        return base+y;
    }
};
```

Template Metaprogramming

- ▷ Use C++ templates as a language in itself

Template Metaprogramming

- ▷ Use C++ templates as a language in itself
- ▷ Type specification as pattern matching → Branching (independently of C++'s control structures)

Template Metaprogramming

- ▷ Use C++ templates as a language in itself
- ▷ Type specification as pattern matching → Branching (independently of C++'s control structures)
- ▷ Template used inside template → Recursion

Template Metaprogramming

- Use C++ templates as a language in itself
- Type specification as pattern matching → Branching (independently of C++'s control structures)
- Template used inside template → Recursion
- Branching and Recursion → **Turing complete**

Template Metaprogramming

- ▷ Use C++ templates as a language in itself
- ▷ Type specification as pattern matching → Branching (independently of C++'s control structures)
- ▷ Template used inside template → Recursion
- ▷ Branching and Recursion → **Turing complete**
- ▷ Allows offloading of computation to compile-time

Factorial: normal C++

```
int factorial(int n) {
    if(n < 1){
        return  1;
    }
    else {
        return n*factorial(n-1);
    }
}
```

Factorial: Template metaprogramming

```
template <int n>
struct factorial {
    const static int result =
                    n * factorial<n - 1>::result;
};

template <>
struct factorial<0> {
    const static int result = 1 ;
};
```

- ▷ Types used for pattern matching.
- ▷ Structs used as functions.
- ▷ Most "specific" template chosen first. (cf. Haskell)

TMP in Armadillo

- ▷ Classes for common LA objects: matrix, vector, etc

TMP in Armadillo

- Classes for common LA objects: matrix, vector, etc
- Literal expression in code converted to template tree, e.g.:
  `A*inv(B)` → `Glue<A, Op<B, op_inv>, op_times>`

TMP in Armadillo

- Classes for common LA objects: matrix, vector, etc
- Literal expression in code converted to template tree, e.g.:
  `A*inv(B)` $\rightarrow$ `Glue<A, Op<B, op_inv>, op_times>`
- Infers information on arguments via pattern matching

TMP in Armadillo

- Classes for common LA objects: matrix, vector, etc
- Literal expression in code converted to template tree, e.g.:
  `A*inv(B)` $\rightarrow$ `Glue<A, Op<B, op_inv>, op_times>`
- Infers information on arguments via pattern matching
- User can provide information about objects.
  e.g.: `diagmat(A)` or `trimatu(A)`

TMP in Armadillo

- Classes for common LA objects: matrix, vector, etc
- Literal expression in code converted to template tree, e.g.:
  `A*inv(B)` → `Glue<A, Op<B, op_inv>, op_times>`
- Infers information on arguments via pattern matching
- User can provide information about objects.
  e.g.: `diagmat(A)` or `trimatu(A)`
- Actual calculation delayed until runtime request for results

Simple example: `inv(inv(A))`

- Rationale: should not compute anything; just return `A`
- Inner call to `inv()` returns a `Op<A, op_inv>` object.
- Outter call matches the definition of `inv()` with `Op<A, op_inv>` argument. Returns `A`.

Simple example: `inv(inv(A))`

- Rationale: should not compute anything; just return `A`
- Inner call to `inv()` returns a `Op<A, op_inv>` object.
- Outter call matches the definition of `inv()` with `Op<A, op_inv>` argument. Returns `A`.
- Advantages:
  - Obvious: no actual inversion.
  - Not obvious: multiple-dispatch without the runtime overhead.
  - Allows compiler to get rid of unneeded function calls.

Matrix Chain Multiplication

- Matrix multiplication is associative.
  $(ABC)D = A(BCD) = \dots$
- Order affects the number of operations.

Matrix Chain Multiplication

- Matrix multiplication is associative.
  $(ABC)D = A(BCD) = \dots$
- Order affects the number of operations.
- $A \in \mathbb{R}^{\mathbf{60} \times \mathbf{5}}$, $B \in \mathbb{R}^{\mathbf{5} \times \mathbf{30}}$, $C \in \mathbb{R}^{\mathbf{30} \times \mathbf{10}}$
  - $(AB)C = (60 \cdot 5 \cdot 30) + (60 \cdot 30 \cdot 10) = 27.000$
  - $A(BC) = (5 \cdot 30 \cdot 10) + (60 \cdot 5 \cdot 10) = 4.500$

Matrix Chain Multiplication

- Matrix multiplication is associative.
  $(ABC)D = A(BCD) = \dots$
- Order affects the number of operations.
- $A \in \mathbb{R}^{\mathbf{60 \times 5}}$, $B \in \mathbb{R}^{\mathbf{5 \times 30}}$, $C \in \mathbb{R}^{\mathbf{30 \times 10}}$
  - $(AB)C = (60 \cdot 5 \cdot 30) + (60 \cdot 30 \cdot 10) = 27.000$
  - $A(BC) = (5 \cdot 30 \cdot 10) + (60 \cdot 5 \cdot 10) = 4.500$
- E.g. Matlab: naive left to right ordering
- Complexity: $\mathcal{O}(n^3)$ with dynamic programming
  Smarter approach: $\mathcal{O}(n \cdot log(n))$

Complex example: A*B*C*D

- Instantiated as:
  `Glue<Glue<Glue<A,B,op_times>,C,op_times>,D,op_times>`
- Determines "depth" of expression: $N = 3$
  (amount of operators with same precedence)

Complex example: A*B*C*D

- Instantiated as:
  `Glue<Glue<Glue<A,B,op_times>,C,op_times>,D,op_times>`
- Determines "depth" of expression: $N = 3$
  (amount of operators with same precedence)
- Pattern matches specialized function for $N = 3$ which:
  - Decide $(ABC)D$ vs. $A(BCD)$
  - Heuristic: $\text{cost}(ABC) := \text{columns}(A) \cdot \text{rows}(C)$
  - Recursively calls reordering function with $N = 2$
  - Thus: linear (but sub-optimal) reordering of operations

Complex example: `A*B*C*D`

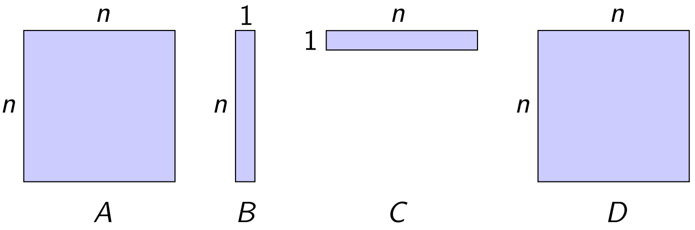
- Instantiated as:
  `Glue<Glue<Glue<A,B,op_times>,C,op_times>,D,op_times>`
- Determines "depth" of expression: $N = 3$
  (amount of operators with same precedence)
- Pattern matches specialized function for $N = 3$ which:
  - Decide $(ABC)D$ vs. $A(BCD)$
  - Heuristic: $\text{cost}(ABC) := \text{columns}(A) \cdot \text{rows}(C)$
  - Recursively calls reordering function with $N = 2$
  - Thus: linear (but sub-optimal) reordering of operations
- Remember: all this at **compile-time**!

Complex example: `A*B*C*D` (cont.)

- $((AB)C)D$ ✓
- $(A(BC))D$ ✓
- $A((BC)D)$ ✓
- $A(B(CD))$ ✓
- $(AB)(CD)$ ✗

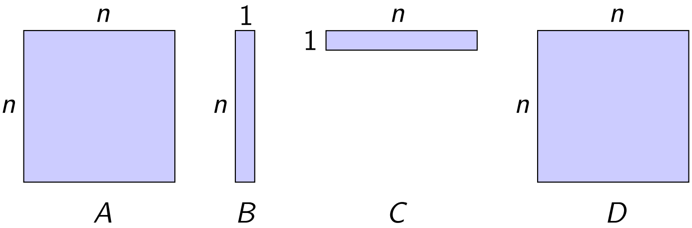
Complex example: `A*B*C*D` (cont.)

- $((AB)C)D$ ✓
- $(A(BC))D$ ✓
- $A((BC)D)$ ✓
- $A(B(CD))$ ✓
- $(AB)(CD)$ ✗



Complex example: `A*B*C*D` (cont.)

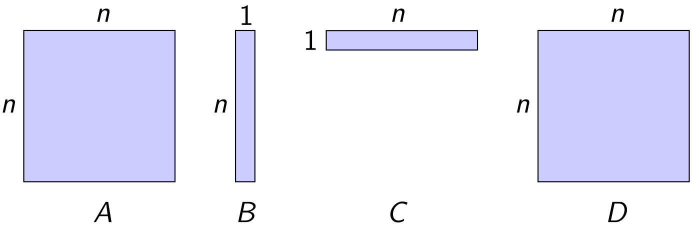
- $((AB)C)D$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $(A(BC))D$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $A((BC)D)$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $A(B(CD))$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $(AB)(CD)$ ✗ $\Rightarrow \mathcal{O}(n^2)$



Complex example: `A*B*C*D` (cont.)

- $((AB)C)D$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $(A(BC))D$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $A((BC)D)$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $A(B(CD))$ ✓ $\Rightarrow \mathcal{O}(n^3)$
- $(AB)(CD)$ ✗ $\Rightarrow \mathcal{O}(n^2)$

n	Armadillo	All
3	2	2
4	4	5
5	8	14
6	16	42
7	32	132



Other improvements

- $A = A_1 + A_2 + \ldots + A_n$
  No temporary matrix besides output

Other improvements

- $A = A_1 + A_2 + \ldots + A_n$
  No temporary matrix besides output
- $y = A^{-1}x \rightarrow y = \mathrm{solve}(A, x)$
  Faster and more accurate

Other improvements

- $A = A_1 + A_2 + \ldots + A_n$
  No temporary matrix besides output
- $y = A^{-1}x \to y = \text{solve}(A, x)$
  Faster and more accurate
- $\text{inv}(\text{diag}(A))$
  Elementwise inverse

Other improvements

- $A = A_1 + A_2 + \ldots + A_n$
  No temporary matrix besides output
- $y = A^{-1}x \rightarrow y = \text{solve}(A, x)$
  Faster and more accurate
- $\text{inv}(\text{diag}(A))$
  Elementwise inverse
- $\text{as\_scalar}(r * X * q)$ with $r \in \mathbb{R}^{1\times n}, X \in \mathbb{R}^{n\times n}, q \in \mathbb{R}^{n\times 1}$
  Result of computation is 1×1 matrix

Other improvements

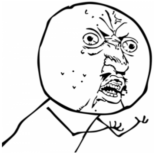
- $A = A_1 + A_2 + \ldots + A_n$
  No temporary matrix besides output
- $y = A^{-1}x \rightarrow y = \mathrm{solve}(A, x)$
  Faster and more accurate
- $\mathrm{inv}(\mathrm{diag}(A))$
  Elementwise inverse
- $\mathrm{as\_scalar}(r * X * q)$ with $r \in \mathbb{R}^{1\times n}, X \in \mathbb{R}^{n\times n}, q \in \mathbb{R}^{n\times 1}$
  Result of computation is 1×1 matrix
- Lots more!

Conclusion

- ✓ Easy to use for non-LA-experts
- ✓ Some optimization comes built-in

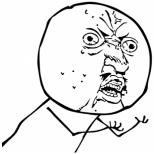
Conclusion

- ✓ Easy to use for non-LA-experts
- ✓ Some optimization comes built-in
- ✗ Cannot optimize all cases
- ✗ Debugging for more complex code can be a problem (C++)



Conclusion

- ✓ Easy to use for non-LA-experts
- ✓ Some optimization comes built-in
- ✗ Cannot optimize all cases
- ✗ Debugging for more complex code can be a problem (C++)



- ▷ Our recommendation: Use it! Don't try to understand it.

Thanks! Questions?

Sources:

- Conrad Sanderson, *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*, Technical Report, NICTA, 2010.
- Armadillo source code.
- *C++ Programming*, chapter *Template Meta-Programming*, Wikibooks
- *Matrix chain multiplication*, Wikipedia