Armadillo: C++ Template Metaprogramming for Compile-Time Optimization of Linear Algebra

Conrad Sanderson, Ryan Curtin

Abstract

Armadillo is a C++ library for linear algebra (matrix maths), enabling mathematical operations to be written in a manner similar to the widely used Matlab language, while at the same time providing efficient computation. It is used for relatively quick conversion of research code into production environments, and as a base for software such as MLPACK (a machine learning library). Through judicious use of template metaprogramming and operator overloading, linear algebra operations are automatically expressed as types, enabling compile-time parsing and evaluation of compound mathematical expressions. This in turn allows the compiler to produce specialized and optimized code. In this talk we cover the fundamentals of template metaprogramming that make compile-time optimizations possible, and demonstrate the internal workings of the Armadillo library. We show that the code automatically generated via Armadillo is comparable with custom C implementations. Armadillo is open source software, available at http://arma.sourceforge.net

Presented at the Platform for Advanced Scientific Computing (PASC) Conference, Switzerland, 2017.

References

[1] C. Sanderson, R. Curtin. *Armadillo: a template-based C++ library for linear algebra.* Journal of Open Source Software, Vol. 1, pp. 26, 2016.

[2] D. Eddelbuettel, C. Sanderson. *RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra.* Computational Statistics and Data Analysis, Vol. 71, 2014.

[3] R. Curtin et al. *MLPACK: A Scalable C++ Machine Learning Library.* Journal of Machine Learning Research, Vol. 14, pp. 801-805, 2013.

[4] C. Sanderson. *An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical Report, NICTA, 2010.

Armadillo: C++ Template Metaprogramming for Compile-Time Optimization of Linear Algebra

Conrad Sanderson, **Ryan R. Curtin**

Arroyo
consortium

Introduction

This talk is about two words: **fast** and **clean**.

Introduction

This talk is about two words: **fast** and **clean**.*

One extreme

Generally-accepted wisdom is that fast code tends to be ugly.

```
    .text
    .align 32                           # 1. function entry alignment
    .globl matrixMultiplyASM            #    (for a faster call)
    .type matrixMultiplyASM, @function
matrixMultiplyASM:
    movaps   (%rdi), %xmm0
    movaps 16(%rdi), %xmm1
    movaps 32(%rdi), %xmm2
    movaps 48(%rdi), %xmm3
    movq $48, %rcx                      # 2. loop reversal
1:                                      #    (for simpler exit condition)
    movss (%rsi, %rcx), %xmm4           # 3. extended address operands
    shufps $0, %xmm4, %xmm4             #    (faster than pointer calculation)
    mulps %xmm0, %xmm4
    movaps %xmm4, %xmm5
    movss 4(%rsi, %rcx), %xmm4
    shufps $0, %xmm4, %xmm4
    mulps %xmm1, %xmm4
    addps %xmm4, %xmm5
    movss 8(%rsi, %rcx), %xmm4
    shufps $0, %xmm4, %xmm4
    mulps %xmm2, %xmm4
    addps %xmm4, %xmm5
    movss 12(%rsi, %rcx), %xmm4
    shufps $0, %xmm4, %xmm4
    mulps %xmm3, %xmm4
    addps %xmm4, %xmm5
    movaps %xmm5, (%rdx, %rcx)
    subq $16, %rcx                      # one 'sub' (vs 'add' & 'cmp')
    jge 1b                               # SF=OF, idiom: jump if positive
    ret
```

The other extreme

Generally-accepted wisdom is also that you need to pay at runtime for clean code.

The other extreme

Generally-accepted wisdom is also that you need to pay at runtime for clean code.

```
X = get_matrix();
Y = X' * X;
```

The other extreme

Generally-accepted wisdom is also that you need to pay at runtime for clean code.

```
X = get_matrix();
Y = X' * X;
```

It’s easy to just call the ASM... but what about more complex expressions that aren’t easily optimized at lower levels?

The other extreme

Generally-accepted wisdom is also that you need to pay at runtime for clean code.

```
X = get_matrix();
Y = X' * X;
```

It's easy to just call the ASM... but what about more complex expressions that aren't easily optimized at lower levels?

```
X = get_matrix();
Y = X(:, 1:10) * X(2:11, :)';
```

The other extreme

Generally-accepted wisdom is also that you need to pay at runtime for clean code.

```
X = get_matrix();
Y = X' * X;
```

It's easy to just call the ASM... but what about more complex expressions that aren't easily optimized at lower levels?

```
X = get_matrix();
Y = X(:, 1:10) * X(2:11, :)';
```

```
X = get_matrix();
Z = get_other_matrix();
Y = X(:, 1:10) * X(2:11, :)' + (Z + 2 * X);
```

The overarching question

- Higher-level languages are typically easy to write code but the code may be slow without further effort.

The overarching question

- Higher-level languages are typically easy to write code but the code may be slow without further effort.
- Lower-level languages typically produce fast code, but it may take a long time to write that code.

The overarching question

- Higher-level languages are typically easy to write code but the code may be slow without further effort.
- Lower-level languages typically produce fast code, but it may take a long time to write that code.
- How can we obtain the best of both worlds?

The overarching question

- Higher-level languages are typically easy to write code but the code may be slow without further effort.
- Lower-level languages typically produce fast code, but it may take a long time to write that code.
- How can we obtain the best of both worlds?

Our approach: use C++'s generic programming facilities in order to provide a clean API that wraps very fast code.

Armadillo: a C++ library for linear algebra

`http://arma.sourceforge.net/`

Armadillo is an open-source linear algebra library in C++ aimed at speed and ease of use.

- Wraps underlying LAPACK/BLAS implementation (or MKL too)
- Uses template metaprogramming framework to avoid unnecessary operations
- Has sparse matrix support
- Has parallelism support internally via OpenMP (or use OpenBLAS)
- Supports 1D, 2D, and 3D objects
- Supports numerous matrix decompositions and other utility functions
- Provides high-level syntax deliberately similar to MATLAB/Octave to ease conversion of research code into production

Adding matrices

Consider the following expression in MATLAB:

```
% x and y are some matrices
z = 2 * (x' + y) + 2 * (x + y');
```

Adding matrices

Consider the following expression in MATLAB:

```
% x and y are some matrices
z = 2 * (x' + y) + 2 * (x + y');
```

What happens?

- `x'` into temporary
- `y'` into temporary
- add everything into output matrix

Adding matrices

Consider the following expression in MATLAB:

```
% x and y are some matrices
z = 2 * (x' + y) + 2 * (x + y');
```

What happens?

- `x'` into temporary
- `y'` into temporary
- add everything into output matrix

This is inefficient especially when `x` and `y` are large!

A faster way

The same operation in C:

A faster way

The same operation in C:

```
void operation(double** z, double** x, double** y, size_t n)
{
  // huge number of possibilities for optimization... this
  // implementation is optimized for slides (space-optimized)
  for (size_t i = 0; i < n; ++i)
    for (size_t j = 0; j < n; ++j)
      z[i][j] = 2 * (x[j][i] + y[i][j]) + \
                2 * (x[i][j] + y[j][i]);
}
```

A faster way

The same operation in C:

```
void operation(double** z, double** x, double** y, size_t n)
{
  // huge number of possibilities for optimization... this
  // implementation is optimized for slides (space-optimized)
  for (size_t i = 0; i < n; ++i)
    for (size_t j = 0; j < n; ++j)
      z[i][j] = 2 * (x[j][i] + y[i][j]) + \
                2 * (x[i][j] + y[j][i]);
}
```

No temporary copies are needed.

A faster way

The same operation in C:

```
void operation(double** z, double** x, double** y, size_t n)
{
  // huge number of possibilities for optimization... this
  // implementation is optimized for slides (space-optimized)
  for (size_t i = 0; i < n; ++i)
    for (size_t j = 0; j < n; ++j)
      z[i][j] = 2 * (x[j][i] + y[i][j]) + \
                2 * (x[i][j] + y[j][i]);
}
```

No temporary copies are needed. **But for every complex expression we have to reimplement the method! This approach doesn't scale.**

Let’s do it in C++!

Let’s restrict ourselves to considering matrix addition for simplicity.

The first thing we’ll need will be some matrix class...

```
class mat
{
 public:
  mat(size_t n_rows, size_t n_cols); // constructor

  double* mem; // the actual matrix memory
  size_t n_rows, n_cols; // the size of the matrix

  mat operator+(const mat& other);
  mat operator=(const mat& other);
};
```

operator+()

```
mat mat::operator+(const mat& other)
{
  mat output(n_rows, n_cols);

  for (size_t i = 0; i < n_cols; ++i)
    for (size_t j = 0; j < n_rows; ++j)
      output.mem[i * n_rows + j] = mem[i * n_rows + j] +
                                   other.mem[i * n_rows + j];

  return output;
}
```

So now what happens?

What happens if we write a simple matrix addition expression?

```
extern mat a, b, c, d; // these are already ready...


mat z = a + b + c + d;
```

So now what happens?

What happens if we write the same expression?

```
extern mat a, b, c, d; // these are already ready...

mat z = a + b + c + d;
```

Code is readable... but horribly slow! Each `operator+()` and `operator=()` incur a copy! This is even worse than MATLAB from earlier.

Op<> **class**

Let's define an auxiliary placeholder type for an operation:

```
template<typename T1, typename T2>
struct op
{
  op(const T1& x, const T2& y): x(x), y(y) { }

  const T1& x;
  const T2& y;
};
```

Op<> **class**

Let's define an auxiliary placeholder type for an operation:

```
template<typename T1, typename T2>
struct op
{
  op(const T1& x, const T2& y): x(x), y(y) { }

  const T1& x;
  const T2& y;
};

template<typename T1, typename T2>
const op<T1, T2> operator+(const T1& x, const T2& y)
{
  return op<T1, T2>(x, y);
}
```

Op<> class

Let's define an auxiliary placeholder type for an operation:

```
template<typename T1, typename T2>
struct op
{
  op(const T1& x, const T2& y): x(x), y(y) { }

  const T1& x;
  const T2& y;
};

template<typename T1, typename T2>
const op<T1, T2> operator+(const T1& x, const T2& y)
{
  return op<T1, T2>(x, y);
}
```

This is a placeholder type that represents that an addition operation needs to be done.

unwrap_elem<>

We also need some utility functions.

```
template<typename T1, typename T2>
inline
double unwrap_elem(const T1& x, const T2& y,
                   int row, int col);
```

We'll specialize this template for some cases...

unwrap_elem<>

We also need some utility functions.

```
template<>
inline
double unwrap_elem(const mat& x, const mat& y,
                   int row, int col)
{
  return x[col * x.n_rows + row] +
         y[col * y.n_rows + row];
}
```

Two matrices: add the elements.

unwrap_elem<>

We also need some utility functions.

```
template<typename T1, typename T2>
inline
double unwrap_elem(const op<T1, T2>& x, const mat& y,
                   int row, int col)
{
  return unwrap_elem(x.x, x.y, row, col) +
         y.mem[col * y.n_rows + row];
}
```

An op and a matrix: call `unwrap_elem<>()` recursively on the op, add the matrix element.

unwrap_elem<>

We also need some utility functions.

```
template<typename T1, typename T2>
inline
double unwrap_elem(const mat& x, const op<T1, T2>& y,
                   int row, int col)
{
  return x.mem[col * x.n_rows + row] +
         unwrap_elem(y.x, y.y, row, col);
}
```

A matrix and an op: add the matrix element and the recursive result of `unwrap_elem<>()` on the op.

unwrap_elem<>

We also need some utility functions.

```
template<typename T1, typename T2, typename T3, typename T4>
inline
double unwrap_elem(const op<T1, T2>& x, const op<T3, T4>& y,
                   int row, int col)
{
  return unwrap_elem(x.x, x.y, row, col) +
         unwrap_elem(y.x, y.y, row, col);
}
```

Two ops: add the recursive results of `unwrap_elem<>()` on each op.

Unwrapping the expression

Now we need `mat` to be able to accept `op<...>` types.

```
template<typename T1, typename T2>
void mat::operator=(const op<T1, T2>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = unwrap_elem(op.x, op.y, r, c);
}
```

Putting it all together...

What types do these expressions return?

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`

  $\rightarrow$ `op<mat, mat>`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`
  $\rightarrow$ `op<mat, mat>`
- `mat + mat + mat`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`

  $\rightarrow$ `op<mat, mat>`

- `mat + mat + mat`

  $\rightarrow$ `op<mat, mat> + mat`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`

  $\rightarrow$ `op<mat, mat>`

- `mat + mat + mat`

  $\rightarrow$ `op<mat, mat> + mat`

  $\rightarrow$ `op<op<mat, mat>, mat>`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`

  $\rightarrow$ `op<mat, mat>`

- `mat + mat + mat`

  $\rightarrow$ `op<mat, mat> + mat`

  $\rightarrow$ `op<op<mat, mat>, mat>`

- `(mat + mat) + (mat + mat)`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`
  $\rightarrow$ `op<mat, mat>`
- `mat + mat + mat`
  $\rightarrow$ `op<mat, mat> + mat`
  $\rightarrow$ `op<op<mat, mat>, mat>`
- `(mat + mat) + (mat + mat)`
  $\rightarrow$ `op<mat, mat> + op<mat, mat>`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`
  $\rightarrow$ `op<mat, mat>`
- `mat + mat + mat`
  $\rightarrow$ `op<mat, mat> + mat`
  $\rightarrow$ `op<op<mat, mat>, mat>`
- `(mat + mat) + (mat + mat)`
  $\rightarrow$ `op<mat, mat> + op<mat, mat>`
  $\rightarrow$ `op<op<mat, mat>, op<mat, mat> >`

Putting it all together...

What types do these expressions return?

```
template<typename T1, T2>
const op<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`
  $\rightarrow$ `op<mat, mat>`
- `mat + mat + mat`
  $\rightarrow$ `op<mat, mat> + mat`
  $\rightarrow$ `op<op<mat, mat>, mat>`
- `(mat + mat) + (mat + mat)`
  $\rightarrow$ `op<mat, mat> + op<mat, mat>`
  $\rightarrow$ `op<op<mat, mat>, op<mat, mat> >`
- and so forth...

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
template<typename T1, typename T2>
void mat::operator=(const op<T1, T2>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = unwrap_elem(op.x, op.y, r, c);
}
```

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op<op<mat, mat>, mat>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = unwrap_elem(op.x, op.y, r, c);
}
```

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op<op<mat, mat>, mat>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = unwrap_elem(op.x, op.y, r, c) ;
}
```

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op<op<mat, mat>, mat>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = (unwrap_elem(op.x.x, op.x.y, r, c) +
            op.y.mem[c * n_rows + r]);
}
```

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op<op<mat, mat>, mat>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = ( (op.x.x.mem[c * n_rows + r] +
                               op.x.y.mem[c * n_rows + r]) +
              op.y.mem[c * n_rows + r]);
}
```

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op<op<mat, mat>, mat>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = (( a .mem[c * n_rows + r] +
                               b .mem[c * n_rows + r]) +
          c [c * n_rows + r]);
}
```

Taking it all apart...

So we have some expression like `z = a + b + c` which yields a type `op<op<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op<op<mat, mat>, mat>& op)
{
  for (size_t c = 0; c < n_cols; ++c)
    for (size_t r = 0; r < n_rows; ++r)
      mem[c * n_rows + r] = (( a .mem[c * n_rows + r] +
                               b .mem[c * n_rows + r]) +
            c [c * n_rows + r]);
}
```

Thus the compiler is generating the fast and efficient code that we want, and we get to preserve our clean syntax!
Armadillo is built on the same basic idea as this example.

What can we do?

By capturing the expression as a compile-time type, a whole host of optimizations are available:

- Avoiding temporary matrix generation (i.e. `a + b.t()`)
- Elementwise operation generation
- Optimized multiplication with special matrix types (diagonal, triangular, etc.)
- Minimal evaluation of expressions like `trace(a * b.t())`
- Allocation-free handling of generated matrices (`ones()`, `zeros()`, `eye()`, etc.)
- Compile-time size checks on fixed-size matrices
- Specialized solvers for triangular matrices

Since all of this is done at compile-time, the compiler can make many additional optimizations, resulting in large speed gains!

Some examples of Armadillo

Here are some examples of some computations in Armadillo, in C++:

```
// Non-negative matrix factorization update rules.
// Schur product (%) is elementwise multiplication.
W = (W % (V * H.t())) / (W * H * H.t());
H = (H % (W.t() * V)) / (W.t() * W * H);

// Linear regression: we want to solve 'r = p * X' for p, so we
// perform QR decomposition of X, then solve for p using r.
mat Q, R;
qr(Q, R, data.t());
solve(parameters /* output */, R, (responses * Q).t());

// Multiply with the first column of a larger matrix.
vec output = matrixOne * matrixTwo.col(0).t();
```

Benchmarks

Task 1: $z = 2(x' + y) + 2(x + y')$.

```
extern int n;
mat x(n, n, fill::randu);
mat y(n, n, fill::randu);
mat z = 2 * (x.t() + y) + 2 * (x + y.t()); // only time this line
```

n	**arma**	**numpy**	**octave**	**R**
1000	**0.029s**	0.040s	0.036s	0.052s
3000	**0.047s**	0.432s	0.376s	0.344s
10000	**0.968s**	5.948s	3.989s	4.952s
30000	**19.167s**	62.748s	41.356s	*fail*

Benchmarks

Task 2: $z = (x + 10 * I)^{\dagger} - y$.

```
extern int n;
mat x(n, n, fill::randu);
mat y(n, n, fill::randu);
mat z = pinv(x + 10 * eye(n, n)) - y; // only time this line
```

n	**arma**	**numpy**	**octave**	**R**
300	**0.081s**	**0.080s**	0.324s	0.096s
1000	**1.321s**	1.354s	26.156s	1.444s
3000	**28.817s**	28.955s	648.64s	29.732s
10000	**777.55s**	785.58s	17661.9s	787.201s

The computation is dominated by the calculation of the pseudoinverse.

Benchmarks

Task 3: $z = abcd$ for decreasing-size matrices.

```
extern int n;
mat a(n, 0.8 * n, fill::randu);
mat b(0.8 * n, 0.6 * n, fill::randu);
mat c(0.6 * n, 0.4 * n, fill::randu);
mat d(0.4 * n, 0.2 * n, fill::randu);
mat z = a * b * c * d; // only time this line
```

n	**arma**	**numpy**	**octave**	**R**
1000	0.042s	0.051s	**0.033s**	0.056s
3000	**0.642s**	0.812s	0.796s	0.846s
10000	**16.320s**	26.815s	26.478s	26.957s
30000	**329.87s**	708.16s	706.10s	707.12s

Armadillo can automatically select the correct ordering for multiplication.

Benchmarks

Task 4: $z = a'(\mathrm{diag}(b)^{-1})c$.

```
extern int n;
vec a(n, fill::randu);
vec b(n, fill::randu);
vec c(n, fill::randu);
double z = as_scalar(a.t() * inv(diagmat(b)) * c); // only time this line
```

n	**arma**	**numpy**	**octave**	**R**
1k	**8e-6s**	0.100s	2e-4s	0.014s
10k	**8e-5s**	49.399s	4e-4s	0.208s
100k	**8e-4s**	*fail*	0.002s	*fail*
1M	0.009s	*fail*	0.024s	*fail*
10M	0.088s	*fail*	0.205s	*fail*
100M	0.793s	*fail*	1.972s	*fail*
1B	8.054s	*fail*	19.520s	*fail*

Related projects

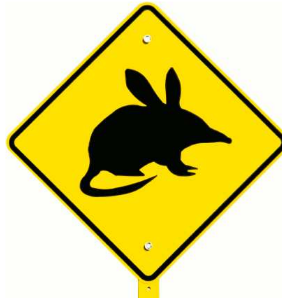
Many projects have been built on top of Armadillo:

- **mlpack**: machine learning library (http://www.mlpack.org)
  Implements standard and cutting-edge machine learning algorithms
  Neural network support in next major release
- **SigPack**: signal processing library (https://sourceforge.net/projects/sigpack/)
- **libpca**: principal components analysis library (http://sourceforge.net/projects/libpca/)
- **matlab2cpp**: Matlab to Armadillo converter (https://github.com/emc2norway/m2cpp)
- **RcppArmadillo**: R-Armadillo bridge (http://cran.r-project.org/web/packages/RcppArmadillo/)
- **armanpy**: Armadillo NumPy bindings (http://sourceforge.net/projects/armanpy/)
- **ERKALE**: quantum chemistry (http://code.google.com/p/erkale/)
- **libdynamica**: physics numerical methods (http://code.google.com/p/libdynamica/)

450+ citations in academic papers.

Future work

Armadillo-like library for GPU matrix operations: **Bandicoot**



`http://coot.sourceforge.io/`

Two separate use case options:

- Bandicoot can be used as a drop-in accelerator to Armadillo, offloading intensive computations to the GPU when possible.
- Bandicoot can be used as its own library for GPU matrix programming.

Future work

Armadillo-like library for GPU matrix operations: **Bandicoot**

```
using namespace coot;
mat x(n, n, fill::randu); // matrix allocated on GPU
mat y(n, n, fill::randu);

mat z = x * y; // computation done on GPU
```

- Bandicoot can be used as its own library for GPU matrix programming.

Conclusion

We can have the best of both worlds using C++ and templates:
high-level, easy-to-prototype code and efficiency.

`http://arma.sourceforge.net/`

References

[1] C. Sanderson, R. Curtin. *Armadillo: a template-based C++ library for linear algebra.* Journal of Open Source Software, Vol. 1, pp. 26, 2016.

[2] D. Eddelbuettel, C. Sanderson. *RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra.* Computational Statistics and Data Analysis, Vol. 71, 2014.

[3] R. Curtin et al. *MLPACK: A Scalable C++ Machine Learning Library.* Journal of Machine Learning Research, Vol. 14, pp. 801-805, 2013.

[4] C. Sanderson. *An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical Report, NICTA, 2010.