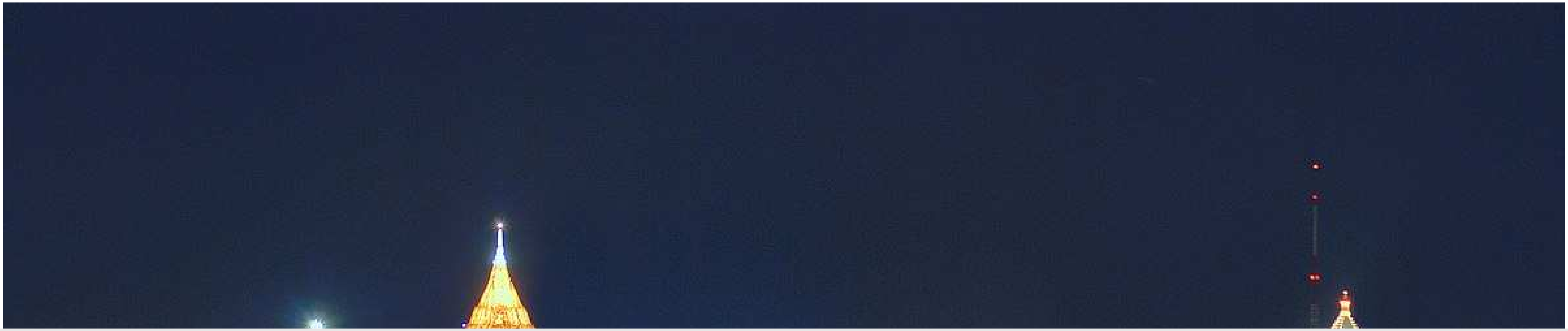
C++ Template Metaprogramming for Compile-Time Optimization of CPU and GPU Linear Algebra

Ryan R. Curtin

Feb. 10, 2026

Geographical Trivia *(Atlanta)*

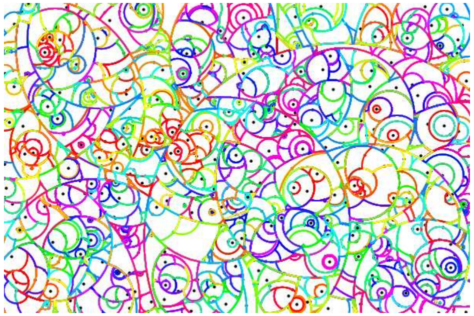
Geographical Trivia *(Atlanta)*



Atlanta quick tips:

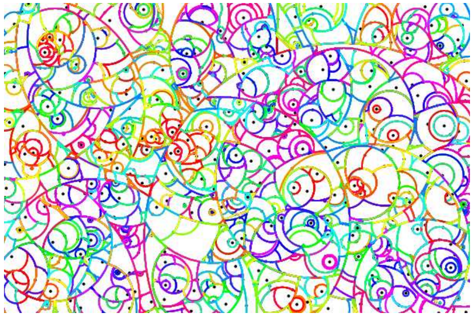
- Atlanta is not good at cold weather events
- The airport is big enough to be the city but it’s not the main city
- Be hungry
- Ride your bike and walk on the Beltline and all its trails
- Check out the Laser Light Show at Stone Mountain and bring lots of cheap beer

Biographical Trivia *(Geometric algorithms)*



Blah blah, fast algorithms, efficient implementations, trees, etc.

Biographical Trivia *(Geometric algorithms)*



Blah blah, fast algorithms, efficient implementations, trees, etc.

Then I became a full-time scientific open-source software developer...

We Are Not Alone

Marcus Edel

Collabora, Inc.

Omar Shrit

Renesas

Conrad Sanderson

Data61 / Griffith University

Dirk Eddelbuettel

UIUC

(plus 250+ other contributors from all over the world!)

Introduction To The Rabbit Hole

My rabbit hole of choice is about two words: **fast** and **clean**.

It had two motivations:

- I need my machine learning code to be fast.
- I need my machine learning code to be long-term maintainable (thus readable, etc.).

This drives us to C++.

C++ primer

C++ is important because we get a separation of **compile time** and **runtime**.

- **Compile time**: `g++ -O3 -o prog prog.cpp -larmadillo`
- **Runtime**: `./prog`

Templates allow us to force the compiler to generate code at **compile time**:

```
template<typename T>
T add(const T& a, const T& b)
{
  // The compiler will generate code for any type T
  // that we called add() with.
  return a + b;
}
```

Template Specialization

We can *specialize* a template function so it has different behavior depending on type...

```
template<typename T>
T add(const T& a, const T& b)
{
  // The compiler will generate code for any type T
  // that we called add() with.
  return a + b;
}

template<>
bool add(const bool& a, const bool& b)
{
  std::cout << "this is a completely different code path!\n";
  return true; // why not?
}
```

This is like a compile-time ‘if’... **how can we abuse this???**

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 (old!) MATLAB:

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

Adding matrices

Consider the following expression in (old!) 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 (old!) 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:

```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:

```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:

```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 a simple matrix addition 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.

We have to turn to template metaprogramming to get what we want...

Expression templates: Veldhuizen, 1996-1998, the Blitz++ library.

Op<> **class**

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

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

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

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

`Op<>` class

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

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

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

template<typename T1, typename T2>
const op_add<T1, T2> operator+(const T1& x, const T2& y)
{
  return op_add<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>
inline
double get_elem(T1& x, int row, int col);
```

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

unwrap_elem<>

We also need some utility functions.

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

For a matrix argument just return the value.

`unwrap_elem<>`

We also need some utility functions.

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

An op: call `get_elem<>()` recursively on both operands.

Unwrapping the expression

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

```
template<typename T1, typename T2>
void mat::operator=(const op_add<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] = get_elem(op, 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_add<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_add<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_add<T1, T2> operator+(const T1& x, const T2& y);
```

- `mat + mat`

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

Putting it all together...

What types do these expressions return?

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

- `mat + mat`

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

Putting it all together...

What types do these expressions return?

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

- `mat + mat`

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

- `mat + mat + mat`

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

Putting it all together...

What types do these expressions return?

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

- `mat + mat`

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

- `mat + mat + mat`

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

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

Putting it all together...

What types do these expressions return?

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

- `mat + mat`

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

- `mat + mat + mat`

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

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

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

Putting it all together...

What types do these expressions return?

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

- `mat + mat`

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

- `mat + mat + mat`

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

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

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

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

Putting it all together...

What types do these expressions return?

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

- `mat + mat`

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

- `mat + mat + mat`

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

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

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

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

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

Putting it all together...

What types do these expressions return?

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

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

Taking it all apart...

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

```
template<typename T1, typename T2>
void mat::operator=(const op_add<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] = get_elem(op, r, c);
}
```

Taking it all apart...

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

```
//
void mat::operator=(const op_add<op_add<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] = get_elem(op, r, c);
}
```

Taking it all apart...

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

```
//
void mat::operator=(const op_add<op_add<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] = get_elem(op, r, c);
}
```

Taking it all apart...

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

```
//
void mat::operator=(const op_add<op_add<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] = get_elem(op.x, r, c) +
                            get_elem(op.y, r, c) ;
}
```

Taking it all apart...

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

```
//
void mat::operator=(const op_add<op_add<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] = get_elem(op.x, r, c) +
                            op.y.mem[c * op.y.n_rows + r];
}
```

Taking it all apart...

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

```
//
void mat::operator=(const op_add<op_add<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] = (get_elem(op.x.x, r, c) +
                             get_elem(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_add<op_add<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op_add<op_add<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_add<op_add<mat, mat>, mat>` that gets `mat::operator=()` called on it... what happens?

```
//
void mat::operator=(const op_add<op_add<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.mem[c * n_rows + r];
}
```

Taking it all apart...

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

```
//
void mat::operator=(const op_add<op_add<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.mem[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.

Let’s get a little crazy

Now what if we want to transpose matrices too?

```
template<typename T1>
struct op_trans
{
  op_trans(T1& x) : x(x) { }

  T1& x;
};
```

Let's get a little crazy

Now what if we want to transpose matrices too?

```
template<typename T1>
struct op_trans
{
  op_trans(T1& x) : x(x) { }

  T1& x;
};

op_trans<mat> mat::t()
{
  return op_trans(*this);
}
```

Just one more function...

Now we need an overload of `get_elem()`...

```
template<typename T1>
inline
double get_elem(op_trans<T1>& op, int r, int c)
{
  return get_elem(op.x, c, r); // transpose!
}
```

Now we are assuming square matrices, but remember, this is slide code, not production code...

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<typename T1, typename T2>
void mat::operator=(const op_add<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] = get_elem(op, r, c);
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=( const op_add<op_add<mat, op_trans<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] = get_elem(op, r, c);
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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] = get_elem(op, r, c);
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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] = get_elem(op.x, r, c) +
                            op.y.mem[c * n_rows + r];
}
```

Now how can we use it?

Let’s make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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] = get_elem(op.x.x, r, c) +
                            get_elem(op.x.y, r, c) +
                            op.y.mem[c * n_rows + r]
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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] +
                            get_elem(op.x.y, r, c) +
                            op.y.mem[c * n_rows + r]
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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] +
                            get_elem(op.x.y, r, c) +
                            op.y.mem[c * n_rows + r]
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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] +
                            get_elem(op.x.y.x, c, r) +
                            op.y.mem[c * n_rows + r]
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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.x.mem[r * n_rows + c] +
                            op.y.mem[c * n_rows + r]
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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[r * n_rows + c] +
                             c.mem[c * n_rows + r]
}
```

Now how can we use it?

Let's make our expression a little more complex: `z = a + b.t() + c`, which yields the type `op_add<op_add<mat, op_trans<mat», mat>`. Here is what the compiler does:

```
template<>
void mat::operator=(const op_add<op_add<mat, op_trans<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[r * n_rows + c] +
                            c.mem[c * n_rows + r]
}
```

Great! Just one more...

Simple matrix multiplication

We are told our users want to multiply matrices too; possibly we should listen to this advice. So:

```
template<typename T1, typename T2>
struct op_mul
{
  op_mul(T1& x, T2& y) : x(x), y(y)
  {
    // multiply into the result
    multiply(x, y, result);
  }

  T1& x;
  T2& y;
  // result matrix temporary, could be optimized another day
  mat result;
};
```

Simple matrix multiplication

We also need a `get_elem()` overload.

```
template<typename T1, typename T2>
double get_elem(op_mul<T1, T2>& x, int r, int c)
{
  return x.result.mem[c * x.result.n_rows + r];
}
```

Simple matrix multiplication

We also need a `get_elem()` overload.

```
template<typename T1, typename T2>
double get_elem(op_mul<T1, T2>& x, int r, int c)
{
  return x.result.mem[c * x.result.n_rows + r];
}
```

And a way to create an `op_mul<>`.

```
template<typename T1, typename T2>
op_mul<T1, T2> operator*(T1& x, T2& y)
{
  return op_mul(x, y);
}
```

Simple matrix multiplication

But, we also need a `multiply()` function, which is used in the `op_mul<>` constructor. This will be not be templated! But there will be four overloads.

Simple matrix multiplication

But, we also need a `multiply()` function, which is used in the `op_mul<>` constructor. This will be not be templated! But there will be four overloads.

```
void multiply(mat& x, mat& y, mat& result)
{
  gemm(x, y, result);
}
```

Two regular matrices: just a regular GEMM call.

Simple matrix multiplication

But, we also need a `multiply()` function, which is used in the `op_mul<>` constructor. This will be not be templated! But there will be four overloads.

```
void multiply(op_trans<mat>& x, mat& y, mat& result)
{
  gemm_transpose_x(x, y, result);
}
```

One transposed matrix: transpose the left input.

Simple matrix multiplication

But, we also need a `multiply()` function, which is used in the `op_mul<>` constructor. This will be not be templated! But there will be four overloads.

```
void multiply(mat& x, op_trans<mat>& y, mat& result)
{
  gemm_transpose_y(x, y, result);
}
```

One transposed matrix: transpose the right input.

Simple matrix multiplication

But, we also need a `multiply()` function, which is used in the `op_mul<>` constructor. This will be not be templated! But there will be four overloads.

```
void multiply(op_trans<mat>& x, op_trans<mat>& y, mat& result)
{
  gemm_transpose_x_and_y(x, y, result);
}
```

Two transposed matrix: call the GEMM that multiplies with two transposed inputs.

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
template<typename T1, typename T2>
void mat::operator=(const op_add<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] = get_elem(op, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                          op_mul<mat, op_trans<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] = get_elem(op, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] = get_elem(op.x, r, c) +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] = get_elem(op.x, r, c) +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] = get_elem(op.x.x, r, c) +
                            get_elem(op.x.y, r, c) +
                            get_elem(op.y, r, c) ;
}
```

Going all in

Here’s a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] = get_elem(op.x.x, r, c) +
                            get_elem(op.x.y, r, c) +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] +
                            get_elem(op.x.y, r, c) +
                            get_elem(op.y, r, c) ;
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] +
                            get_elem(op.x.y, r, c) +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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] +
                            get_elem(op.x.y.x, c, r) +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                  op_mul<mat, op_trans<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.x.mem[r * n_rows + c] +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                   op_mul<mat, op_trans<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.x.mem[r * n_rows + c] +
                            get_elem(op.y, r, c);
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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.x.mem[r * n_rows + c] +
                            op.y.result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                op_mul<mat, op_trans<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.x.mem[r * n_rows + c] +
                            op.y.result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<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[r * n_rows + c] +
                            op.y.result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                op_mul<mat, op_trans<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[r * n_rows + c] +
                            op.y.result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<mat> > >& op)
{
  mat result;
  multiply(op.y.x, op.y.y, result);

  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[r * n_rows + c] +
                            result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<mat> > >& op)
{
  mat result;
  multiply(op.y.x, op.y.y, result);

  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[r * n_rows + c] +
                            result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<mat> > >& op)
{
  mat result;
  gemm_transpose_y(op.y.x, op.y.y, result);

  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[r * n_rows + c] +
                            result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<mat> > >& op)
{
  mat result;
  gemm_transpose_y( c ,  d , result);

  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[r * n_rows + c] +
                            result.mem[c * n_rows + r];
}
```

Going all in

Here's a new expression: `a + b.t() + (c * d.t())`. Its type will be `op_add<op_add<mat, op_trans<mat> >, op_mul<mat, op_trans<mat> > >`.

```
//
void mat::operator=(const op_add<op_add<mat, op_trans<mat> >,
                                 op_mul<mat, op_trans<mat> > >& op)
{
  mat result;
  gemm_transpose_y( c , d , result);

  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[r * n_rows + c] +
                            result.mem[c * n_rows + r];
}
```

We did it!

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**	**Julia**
1000	0.029s	0.040s	0.036s	0.052s	**0.027s**
3000	0.047s	0.432s	0.376s	0.344s	**0.041s**
10000	**0.968s**	5.948s	3.989s	4.952s	3.683s
30000	**19.167s**	62.748s	41.356s	*fail*	36.730s

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**	**Julia**
300	**0.081s**	**0.080s**	0.324s	0.096s	0.098s
1000	1.321s	1.354s	26.156s	1.444s	**1.236s**
3000	**28.817s**	28.955s	648.64s	29.732s	29.069s
10000	**777.55s**	785.58s	17661.9s	787.201s	778.472s

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**	**Julia**
1000	0.042s	0.051s	**0.033s**	0.056s	0.037s
3000	**0.642s**	0.812s	0.796s	0.846s	0.844s
10000	**16.320s**	26.815s	26.478s	26.957s	26.576s
30000	**329.87s**	708.16s	706.10s	707.12s	704.032s

Armadillo can automatically select the correct ordering for multiplication.

Benchmarks

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

```
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**	**Julia**
1k	**8e-6s**	0.100s	2e-4s	0.014s	0.057s
10k	**8e-5s**	49.399s	4e-4s	0.208s	18.189s
100k	**8e-4s**	*fail*	0.002s	*fail*	*fail*
1M	0.009s	*fail*	0.024s	*fail*	*fail*
10M	0.088s	*fail*	0.205s	*fail*	*fail*
100M	0.793s	*fail*	1.972s	*fail*	*fail*
1B	8.054s	*fail*	19.520s	*fail*	*fail*

Related projects

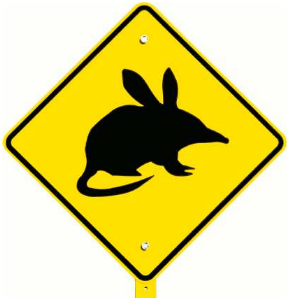
Many projects have been built on top of Armadillo:

- **mlpack**: machine learning library (`http://www.mlpack.org`)
- **ensmallen**: numerical optimization library (`http://www.ensmallen.org`)
- **SigPack**: signal processing library (`https://sourceforge.net/projects/sigpack/`)
- **libpca**: principal components analysis library (`http://sourceforge.net/projects/libpca/`)
- **matlab2cpp**: Matlab–Armadillo converter (`https://github.com/jonathf/m2cpp`) (currently inactive)
- **RcppArmadillo**: R-Armadillo bridge (`http://cran.r-project.org/web/packages/RcppArmadillo/`)
- **ERKALE**: quantum chemistry (`https://github.com/susilehtola/erkale/`)
- ... and so on ...

2000+ citations in academic papers, 30 million+ downloads, used widely in industry, ...

Moving on to GPUs...

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.

Bandicoot API example

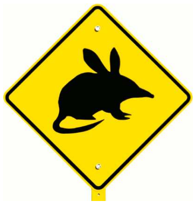
```
using namespace coot;
mat x(n, n, fill::randu); // matrix allocated on GPU
mat y(n, n, fill::randu);

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

Since this is the same as Armadillo, just replace `using namespace arma` with `using namespace coot` (...some caveats apply...)

GPU basics

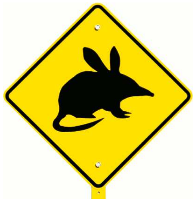
- **SIMD**: all threads execute the same code!
- There are a lot of threads (thousands+)!
- Writing a kernel function is just writing the generic function for each thread.
- Different manufacturers have different implementation languages:
  - NVIDIA / CUDA
  - Apple / Metal
  - AMD / HIP/ROCm
  - Intel / OneAPI
  - OpenCL (not vendor-specific)
  - Vulkan (not vendor-specific)

General Bandicoot design



- We can reuse the template metaprogramming framework ideas from Armadillo.
(e.g. optimize out two transposes; optimize expression order; simplify expressions at compile time based on types)

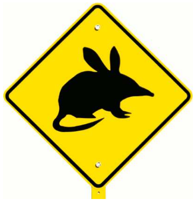
General Bandicoot design



- We can reuse the template metaprogramming framework ideas from Armadillo.
  (e.g. optimize out two transposes; optimize expression order; simplify expressions at compile time based on types)

- We can implement several *backends* for different device types.
  CUDA, OpenCL, Vulkan, Metal, ... *(selectable at compile time or runtime)*

General Bandicoot design

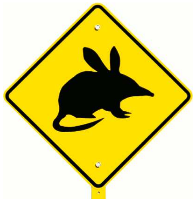


- We can reuse the template metaprogramming framework ideas from Armadillo.
  (e.g. optimize out two transposes; optimize expression order; simplify expressions at compile time based on types)

- We can implement several *backends* for different device types.
  CUDA, OpenCL, Vulkan, Metal, ... *(selectable at compile time or runtime)*

- Compile-time expressions will be unwrapped into a backend call (or a series of them).
  Much like Armadillo maps to LAPACK calls: GEMM, GEMV, etc.

General Bandicoot design



- We can reuse the template metaprogramming framework ideas from Armadillo.
  (e.g. optimize out two transposes; optimize expression order; simplify expressions at compile time based on types)

- We can implement several *backends* for different device types.
  CUDA, OpenCL, Vulkan, Metal, ... *(selectable at compile time or runtime)*

- Compile-time expressions will be unwrapped into a backend call (or a series of them).
  Much like Armadillo maps to LAPACK calls: GEMM, GEMV, etc.

- Backend calls will use custom kernel functions that are just-in-time compiled.
  JIT compilation is necessary because hardware may change between runs.

GPU kernel example

```
// Set every element of the matrix `out` to `val`.
__global__ void fill(float* out_mem,
                     const size_t out_n_rows,
                     const size_t out_n_cols,
                     const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;

  if (row < out_n_rows && col < out_n_cols)
    out_mem[row + col * out_n_rows] = val;
}
```

GPU kernel example

```
// Set every element of the matrix `out` to `val`.
__global__ void fill(float* out_mem,
                     const size_t out_n_rows,
                     const size_t out_n_cols,
                     const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;

  if (row < out_n_rows && col < out_n_cols)
    out_mem[row + col * out_n_rows] = val;
}
```

How can we make this generic? Any type of matrix; any type of expression.

GPU kernel example

```
// Set every element of the column vector `out` to `val`.
__global__ void fill(float* out_mem,
                     const size_t out_n_elem,
                     const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < out_n_elem)
    out_mem[row] = val;
}
```

GPU kernel example

```
// Set every element of the submatrix `out` to `val`.
__global__ void fill(float* out_mem,
                     const size_t out_n_rows,
                     const size_t out_n_cols,
                     const size_t out_leading_dim,
                     const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;

  if (row < out_n_rows && col < out_n_cols)
    out_mem[row + col * out_leading_dim] = val;
}
```

GPU kernel example

```
// Set every element of the 3-D tensor `out` to `val`.
__global__ void fill(float* out_mem,
                     const size_t out_n_rows,
                     const size_t out_n_cols,
                     const size_t out_n_slices,
                     const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (row < out_n_rows && col < out_n_cols && slice < out_n_slices)
    out_mem[row + col * out_n_rows +
            slice * out_n_rows * out_n_cols] = val;
}
```

Kernel generation desiderata

- We should be able to generate a kernel for an expression fully at compile time.

  At runtime, the device driver will compile the kernel (a string).

Kernel generation desiderata

- We should be able to generate a kernel for an expression fully at compile time.

    At runtime, the device driver will compile the kernel (a string).

- We should be able to fuse individual operations together into a single kernel.

    `a+b+c+d` should take one kernel call, not three!

    ...not every operation should be fused together.

Kernel generation desiderata

- We should be able to generate a kernel for an expression fully at compile time.
  - At runtime, the device driver will compile the kernel (a string).

- We should be able to fuse individual operations together into a single kernel.
  - `a+b+c+d` should take one kernel call, not three!
  - ...not every operation should be fused together.

- The inputs and outputs of kernels should be arbitrary Bandicoot objects (not expressions).
  - Matrices, vectors, submatrices, cubes, diagonals...

Overarching goal: condense expressions into as few kernels as possible... at C++ compile time.

Macro-ization...

```
// Set every element of the matrix `out` to `val`.
__global__ void fill(float* out_mem,
                     const size_t out_n_rows,
                     const size_t out_n_cols,
                     const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (row < out_n_rows && col < out_n_cols)
    out_mem[row + col * out_n_rows] = val;
}
```

Macro-ization...

```
#define NAME mf_fill

// Set every element of the matrix `out` to `val`.
__global__ void (NAME)(float* out_mem,
                       const size_t out_n_rows,
                       const size_t out_n_cols,
                       const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (row < out_n_rows && col < out_n_cols)
    out_mem[row + col * out_n_rows] = val;
}
```

Macro-ization...

```
#define NAME mf_fill
#define PARAMS float* out_mem, \
                      const size_t out_n_rows, \
                      const size_t out_n_cols

// Set every element of the matrix `out` to `val`.
__global__ void (NAME)(PARAMS,
                       const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (row < out_n_rows && col < out_n_cols)
    out_mem[row + col * out_n_rows] = val;
}
```

Macro-ization...

```
#define NAME mf_fill
#define PARAMS float* out_mem, \
                      const size_t out_n_rows, \
                      const size_t out_n_cols
#define BOUNDS_CHECK(r, c, s) (r < out_n_rows && c < out_n_cols)

// Set every element of the matrix `out` to `val`.
__global__ void (NAME)(PARAMS,
                       const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (BOUNDS_CHECK(row, col, slice))
    out_mem[row + col * out_n_rows] = val;
}
```

Macro-ization...

```
#define NAME mf_fill
#define PARAMS float* out_mem, \
                      const size_t out_n_rows, \
                      const size_t out_n_cols
#define BOUNDS_CHECK(r, c, s) (r < out_n_rows && c < out_n_cols)
#define AT(r, c, s) out_mem[r + c * out_n_rows]

// Set every element of the matrix `out` to `val`.
__global__ void (NAME)(PARAMS,
                       const float val)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (BOUNDS_CHECK(row, col, slice))
    AT(r, c, s) = val;
}
```

Macros for specific expressions

So the strategy is this:

- At compile-time, generate string definitions of each of the macros we need:
  - Name of the kernel (`NAME`)
  - Parameters for the object (`PARAMS`)
  - Function to perform a bounds check (`BOUNDS_CHECK(r, c, s)`)
  - Function to access an element (`AT(r, c, s)`)
- Pair these definitions with a 'skeleton kernel' that uses the macro.
- Let the device driver compile it and away you go!

Example macros

`PARAMS`

- Matrix:

  `T* mem, size_t n_rows, size_t n_cols`
- Vector:

  `T* mem, size_t n_elem`
- Cube:

  `T* mem, size_t n_rows, size_t n_cols, size_t n_slices`
- Submatrix:

  `T* mem, size_t n_rows, size_t n_cols, size_t leading_dim`

Example macros

AT

- Matrix:

  `mem[r + c * n_rows]`
- Vector:

  `mem[r]`
- Cube:

  `mem[r + c * n_rows + s * n_rows * n_cols]`
- Submatrix:

  `mem[r + c * leading_dim]`

Now what about expressions?

How about... `op_add<mat, mat>`?

```
#define PARAMS double* a_mem, \
               double* b_mem, \
               size_t n_rows, \
               size_t n_cols
```

Now what about expressions?

How about... `op_add<mat, mat>`?

```
#define PARAMS double* a_mem, \
               double* b_mem, \
               size_t n_rows, \
               size_t n_cols

#define BOUNDS_CHECK(r, c, s) (r < n_rows && c < n_cols)
```

Now what about expressions?

How about... `op_add<mat, mat>`?

```
#define PARAMS double* a_mem, \
               double* b_mem, \
               size_t n_rows, \
               size_t n_cols

#define BOUNDS_CHECK(r, c, s) (r < n_rows && c < n_cols)

#define AT(r, c, s) (a_mem[r + c * n_rows] + b_mem[r + c * n_rows])
```

How about another skeleton kernel?

Copy one object to another.

```
// I hid a bit of complexity in the last slide!
// We might need to define different macros for multiple inputs.
__global__ void (NAME)(PARAMS_OUT,
                       PARAMS_IN)
{
  // Determine the thread's position in the matrix.
  const size_t row = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t col = blockIdx.y * blockDim.y + threadIdx.y;
  const size_t slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (BOUNDS_CHECK_OUT(row, col, slice))
    AT_OUT(row, col, slice) = AT_IN(row, col, slice);
}
```

Transposition!

Now... `mat = op_trans<mat>`.

```
#define PARAMS_OUT double* out_mem, \
                   size_t out_n_rows, \
                   size_t out_n_cols
#define PARAMS_IN double* in_mem, \
                  size_t in_n_rows, \
                  size_t in_n_cols
```

Transposition!

Now... `mat = op_trans<mat>`.

```
#define PARAMS_OUT double* out_mem, \
                   size_t out_n_rows, \
                   size_t out_n_cols
#define PARAMS_IN double* in_mem, \
                  size_t in_n_rows, \
                  size_t in_n_cols

#define BOUNDS_CHECK_OUT(r, c, s) (r < out_n_rows && c < out_n_cols)
#define BOUNDS_CHECK_IN(r, c, s) (r < in_n_cols && c < in_n_rows)
```

Transposition!

Now... `mat = op_trans<mat>`.

```
#define PARAMS_OUT double* out_mem, \
                   size_t out_n_rows, \
                   size_t out_n_cols
#define PARAMS_IN double* in_mem, \
                  size_t in_n_rows, \
                  size_t in_n_cols

#define BOUNDS_CHECK_OUT(r, c, s) (r < out_n_rows && c < out_n_cols)
#define BOUNDS_CHECK_IN(r, c, s) (r < in_n_cols && c < in_n_rows)

#define AT_OUT(r, c, s) out_mem[r + c * out_n_rows]
#define AT_IN(r, c, s) in_mem[c + r * out_n_rows]
```

(Aside: this is actually not the best way to do a transpose because of memory access patterns... but it's perfect for slides!)

One more... recursive...

```
mat = op_trans<op_add<mat, mat> >

#define PARAMS_OUT double* out_mem, \
                   size_t out_n_rows, \
                   size_t out_n_cols
#define PARAMS_IN double* in_a_mem, \
                  double* in_b_mem, \
                  size_t in_n_rows, \
                  size_t in_n_cols
```

One more... recursive...

```
mat = op_trans<op_add<mat, mat> >

#define PARAMS_OUT double* out_mem, \
                   size_t out_n_rows, \
                   size_t out_n_cols
#define PARAMS_IN double* in_a_mem, \
                  double* in_b_mem, \
                  size_t in_n_rows, \
                  size_t in_n_cols

#define BOUNDS_CHECK_OUT(r, c, s) (r < out_n_rows && c < out_n_cols)
#define BOUNDS_CHECK_IN(r, c, s) (r < in_n_rows && c < in_n_cols)
```

One more... recursive...

```
mat = op_trans<op_add<mat, mat> >

#define PARAMS_OUT double* out_mem, \
                   size_t out_n_rows, \
                   size_t out_n_cols
#define PARAMS_IN double* in_a_mem, \
                  double* in_b_mem, \
                  size_t in_n_rows, \
                  size_t in_n_cols

#define BOUNDS_CHECK_OUT(r, c, s) (r < out_n_rows && c < out_n_cols)
#define BOUNDS_CHECK_IN(r, c, s) (r < in_n_rows && c < in_n_cols)

#define AT_OUT(r, c, s) out_mem[r + c * out_n_rows]
#define AT_IN(r, c, s) (in_a_mem[c + r * in_n_rows] + \
                        in_b_mem[c + r * in_n_rows])
```

Back To The Higher Level

- This is how kernels in Bandicoot are implemented.
- Individual expressions are optimized into single kernels—where it makes sense to do so.
- Kernels are generated for either the CUDA or OpenCL backend (and soon, Vulkan, Metal, and HIP/ROCm).
- More complex decompositions (SVD, etc.) use adaptations of MAGMA.
- API compatibility with Armadillo is a priority, but not all Armadillo functions are implemented yet.

  ...do you have a particular need? We can prioritize it...

Conclusion

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/`
`http://coot.sourceforge.net/`

R. Curtin, M. Edel, C. Sanderson. “Bandicoot: A Templated C++ Library for GPU Linear Algebra.” The 26th International Conference on Parallel and Distributed Computing, Applications and Technologies (2025).

C. Sanderson, R. Curtin. “Armadillo: a template-based C++ library for linear algebra.” Journal of Open Source Software (2016).

See you at ICMS?