Fast GPU Linear Algebra via Compile Time Expression Fusion

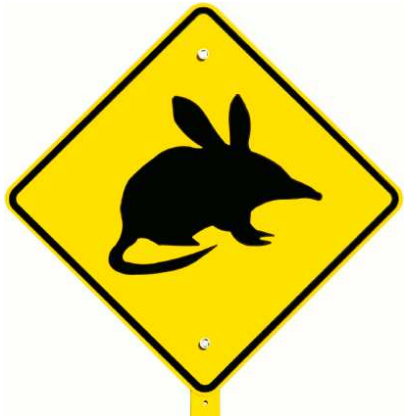
Ryan R. Curtin, Marcus Edel, Conrad Sanderson

July 23, 2026

Introduction

Armadillo

`https://arma.sourceforge.net/`



Bandicoot

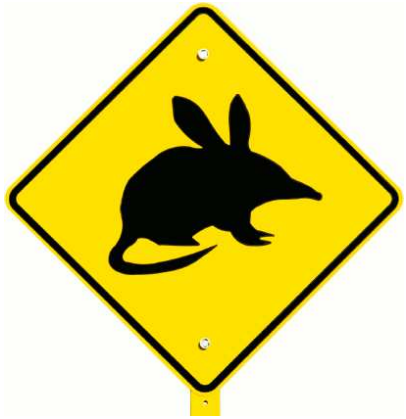
`https://coot.sourceforge.net/`

C++ linear algebra libraries...

Introduction

Armadillo

`https://arma.sourceforge.net/`



Bandicoot

`https://coot.sourceforge.net/`

C++ linear algebra libraries...

- for the CPU (Armadillo) and GPU (Bandicoot),
- matching the intuitive MATLAB API,
- usable directly in C++ or from other languages such as R, MATLAB, etc. via bindings,
- originally aimed at making it easy to get a quick speed boost for your code *(often 10x+!)*

People

Marcus Edel

Collabora, Inc.

Omar Shrit

Renesas

Conrad Sanderson

Data61 / Griffith University

Dirk Eddelbuettel

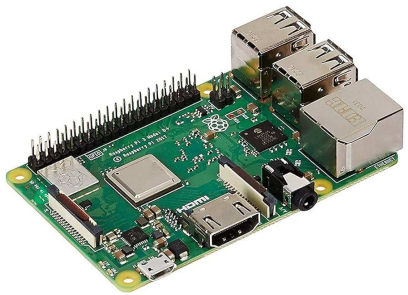
UIUC

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

C++

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

- Compile once, run many times.
- Move computation from runtime to compile-time!
- No runtime environment needed.
- Cross-compilation to other platforms is easy. *(Perhaps less relevant to this community...)*



C++ primer

C++ is important because we get a separation of **compile time** and **runtime**. *What does this look like?*

- **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` based on types... it is all we need to shift computation to compile time.

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

Ideally we want code to run that's a little bit like this:

```
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.

Expression Templates / Delayed Evaluation

With C++, when a user writes an expression like

```
mat Z = 2 * (X + Y.t()) + 2 * (X.t() + Y);
```

Expression Templates / Delayed Evaluation

With C++, when a user writes an expression like

```
mat Z = 2 * (X + Y.t()) + 2 * (X.t() + Y);
```

we can collect the AST (abstract syntax tree) of the expression as a **type**:

```
Op2<
  Op<
    Op2< mat, Op< mat, trans >, op_add >, // X + Y.t()
    op_scalar_times >,                    // 2 * ...
  Op<
    Op2< Op< mat, trans >, mat, op_add >, // X.t() + Y
    op_scalar_times >,                    // 2 * ...
op_add >
```

Expression Templates / Delayed Evaluation

With C++, when a user writes an expression like

```
mat Z = 2 * (X + Y.t()) + 2 * (X.t() + Y);
```

we can collect the AST (abstract syntax tree) of the expression as a **type**:

```
Op2<
  Op<
    Op2< mat, Op< mat, trans >, op_add >, // X + Y.t()
    op_scalar_times >,                    // 2 * ...
  Op<
    Op2< Op< mat, trans >, mat, op_add >, // X.t() + Y
    op_scalar_times >,                    // 2 * ...
op_add >
```

Template specialization can then be used so that the compiler **generates** the desired implementation.

(Note that only the compiler ever encounters the awful type above!)

What can we do?

This idea is called **expression templates** or **delayed evaluation** (Veldhuizen, 1998). 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

Here are some examples of some computations in Armadillo or Bandicoot, 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();
```

Armadillo 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	**0.004s**	0.008s	0.016s	0.016s	0.052s
3000	**0.055s**	0.088s	0.191s	0.121s	0.161s
10000	**1.130s**	1.353s	2.151s	1.632s	1.958s
30000	**11.457s**	15.697s	20.115s	18.806s	17.388s

Benchmarks

Task 2: $z = a'(\mathrm{diag}(b)^{-1})c$.

```
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	Julia
1k	**1.3e-6s**	0.0244s	2.3e-5s	0.0318s	0.0153s
10k	**1.4e-5s**	6.3994s	4.3e-5s	6.5200s	0.3710s
100k	**9.6e-5s**	*fail*	0.0017s	*fail*	*fail*
1M	**0.0015s**	*fail*	0.0069s	*fail*	*fail*
10M	**0.0199s**	*fail*	0.0589s	*fail*	*fail*
100M	**0.2007s**	*fail*	0.5763s	*fail*	*fail*
1B	**2.0149s**	*fail*	5.6637s	*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, ...

Okay but this talk is about GPUs...

How can we get the same speedups from compile-time computation when we are using GPUs?

Okay but this talk is about GPUs...

How can we get the same speedups from compile-time computation when we are using GPUs?

CPU: two stages—compile and run
GPU: three stages—compile host code, JIT-compile GPU kernel, run

- *GPU kernel compilation happens at program runtime*

Okay but this talk is about GPUs...

How can we get the same speedups from compile-time computation when we are using GPUs?

CPU: two stages—compile and run
GPU: three stages—compile host code, JIT-compile GPU kernel, run

- *GPU kernel compilation happens at program runtime*

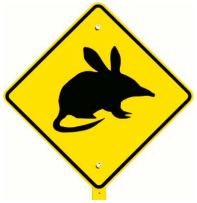
CPU: get the compiler to emit optimized and efficient code
GPU: get the compiler to emit optimized and efficient code
for the GPU that will be compiled by the GPU at runtime

- *may require some runtime code generation*

GPU basics

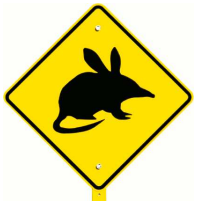
- **SIMD**: all threads execute the same code!
- There are a lot of threads (thousands+)!
- Writing a kernel function is just writing the generic function for each thread.
- Different manufacturers have different implementation languages:
  - NVIDIA / CUDA
  - Apple / Metal
  - AMD / HIP/ROCm
  - Intel / 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)

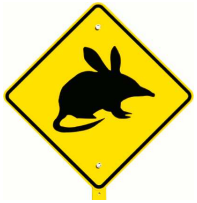
General Bandicoot design



- 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

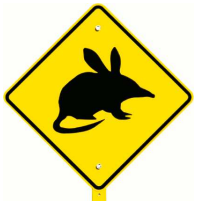


- We can reuse the template metaprogramming framework ideas from Armadillo.
  (e.g. optimize out two transposes; optimize expression order; simplify expressions at compile time based on types)

- 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

```
// Copy the matrix `in` to `out`.  (Assume they are the same size.)
__global__ void copy(float* out_mem,
                     const int out_n_rows,
                     const int out_n_cols,
                     float* in_mem)
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;

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

GPU kernel example

```
// Copy the submatrix of `in` of size `out_n_rows` x `out_n_cols`
// to `out`.
__global__ void copy(float* out_mem,
                     const int out_n_rows,
                     const int out_n_cols,
                     float* in_mem,
                     const int in_leading_dim)
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;

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

GPU kernel example

```
// Copy the transposed submatrix of `in` of size `out_n_rows`
// x `out_n_cols` to `out`.
__global__ void copy(float* out_mem,
                     const int out_n_rows,
                     const int out_n_cols,
                     float* in_mem,
                     const int in_leading_dim)
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;

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

GPU kernel example

```
// Copy the transposed submatrix of `in` of size `out_n_rows`
// x `out_n_cols` to `out`.
__global__ void copy(float* out_mem,
```

If we write these by hand, we cannot hope to compute

```
2 * (X + Y.t()) + 2 * (X.t() + Y)
```

in one kernel call.

We have to generate GPU kernels (preferably at compile time).

```
    out_mem[row + col * out_n_rows] = in_mem[col + row * in_leading_dim];
}
```

Macro-ization...

```
// Copy the matrix `in` to `out`.
__global__ void copy(float* out_mem,
                     const int out_n_rows,
                     const int out_n_cols,
                     float* in_mem)
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;
  const int slice = blockIdx.z * blockDim.z + threadIdx.z;

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

Macro-ization...

```
#define OBJECT_0_PARAMS(out) float* out_mem, \
    const int out_n_rows, const int out_n_cols
#define OBJECT_1_PARAMS(in) float* in_mem, \
    const int in_n_rows, const int in_n_cols

// Copy the object/expression `in` to `out`.
__global__ void copy(OBJECT_0_PARAMS(out),
                     OBJECT_1_PARAMS(in))
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;
  const int slice = blockIdx.z * blockDim.z + threadIdx.z;

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

Macro-ization...

```
#define OBJECT_0_BOUNDS_CHECK(out, row, col) \
    ((row < out_n_rows) && (col < out_n_cols))

// Copy the object/expression `in` to `out`.
__global__ void copy(OBJECT_0_PARAMS(out),
                     OBJECT_1_PARAMS(in))
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;
  const int slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (OBJECT_0_BOUNDS_CHECK(out, row, col))
    out_mem[row + col * out_n_rows] = in[row + col * out_n_rows];
}
```

Macro-ization...

```
#define OBJECT_0_AT(out, row, col) out_mem[row + col * out_n_rows]
#define OBJECT_1_AT(in, row, col) in_mem[row + col * in_n_rows]

// Copy the object/expression `in` to `out`.
__global__ void copy(OBJECT_0_PARAMS(out),
                     OBJECT_1_PARAMS(in))
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;
  const int slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (OBJECT_0_BOUNDS_CHECK(out, row, col))
    OBJECT_0_AT(out, row, col) = OBJECT_1_AT(in, row, col);
}
```

Skeleton kernels

Now **every** Bandicoot operation of the form `lhs = rhs` uses this “skeleton kernel”:

```
// Copy the object/expression `in` to `out`.
__global__ void copy(OBJECT_0_PARAMS(out),
                     OBJECT_1_PARAMS(in))
{
  // Determine the thread's position in the matrix.
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;
  const int slice = blockIdx.z * blockDim.z + threadIdx.z;

  if (OBJECT_0_BOUNDS_CHECK(out, row, col))
    OBJECT_0_AT(out, row, col) = OBJECT_1_AT(in, row, col);
}
```

The macros are set according to what `lhs` and `rhs` are.

Macro examples

```
// out = in; (just a matrix copy)

#define OBJECT_0_PARAMS(out) float* out_mem, \
    const int out_n_rows, const int out_n_cols
#define OBJECT_1_PARAMS(in) float* in_mem, \
    const int in_n_rows, const in in_n_cols

#define OBJECT_0_BOUNDS_CHECK(out, row, col) \
    ((row < out_n_rows) && (col < out_n_cols))

#define OBJECT_0_AT(out, row, col) out_mem[row + col * out_n_rows]
#define OBJECT_1_AT(in, row, col) in_mem[row + col * in_n_rows]
```

Macro examples

```
// out = in.submat(3, 3, 5, 5); (submatrix extraction)

#define OBJECT_0_PARAMS(out) float* out_mem, \
    const int out_n_rows, const int out_n_cols
#define OBJECT_1_PARAMS(in) float* in_mem, \
    const int in_n_rows, const in in_n_cols, const int in_leading_dim

#define OBJECT_0_BOUNDS_CHECK(out, row, col) \
    ((row < out_n_rows) && (col < out_n_cols))

#define OBJECT_0_AT(out, row, col) out_mem[row + col * out_n_rows]
#define OBJECT_1_AT(in, row, col) in_mem[row + col * in_leading_dim]
```

Macro examples

```
// out = in.submat(3, 3, 5, 5).t(); (transposed submatrix extraction)

#define OBJECT_0_PARAMS(out) float* out_mem, \
    const int out_n_rows, const int out_n_cols
#define OBJECT_1_PARAMS(in) float* in_mem, \
    const int in_n_rows, const in in_n_cols, const int in_leading_dim

#define OBJECT_0_BOUNDS_CHECK(out, row, col) \
    ((row < out_n_rows) && (col < out_n_cols))

#define OBJECT_0_AT(out, row, col) out_mem[row + col * out_n_rows]
#define OBJECT_1_AT(in, row, col) in_mem[col + row * in_leading_dim]
```

Macro examples

```
// out = in_A + 2 * in_B; (elementwise addition)

#define OBJECT_0_PARAMS(out) float* out_mem, \
    const int out_n_rows, const int out_n_cols
#define OBJECT_1_PARAMS(in) float* in_A_mem, \
    const int in_A_n_rows, const int in_A_n_cols, \
    float* in_B, const int in_B_n_rows, const int in_B_n_cols, \
    const float in_B_scalar

#define OBJECT_0_BOUNDS_CHECK(out, row, col) \
    ((row < out_n_rows) && (col < out_n_cols))

#define OBJECT_0_AT(out, row, col) out_mem[row + col * out_n_rows]
#define OBJECT_1_AT(in, row, col) in_A_mem[row + col * in_A_n_rows] + \
    in_B_scalar * in_B_mem[row + col * in_B_n_rows]
```

Kernel generation summary

What Bandicoot does:

- Expression templates are used to collect the AST of an expression as a type at compile time.

Kernel generation summary

What Bandicoot does:

- Expression templates are used to collect the AST of an expression as a type at compile time.
- This AST is used at *compile time* to generate definitions of each of the macros above.

Kernel generation summary

What Bandicoot does:

- Expression templates are used to collect the AST of an expression as a type at compile time.
- This AST is used at *compile time* to generate definitions of each of the macros above.
- At program runtime, the macros and skeleton kernel are combined and given to the GPU driver.
  (that is, if the kernel for the operation is not already cached)

Kernel generation summary

What Bandicoot does:

- Expression templates are used to collect the AST of an expression as a type at compile time.
- This AST is used at *compile time* to generate definitions of each of the macros above.
- At program runtime, the macros and skeleton kernel are combined and given to the GPU driver.
  (that is, if the kernel for the operation is not already cached)
- The linear algebra operation then runs (fast!).

Kernel generation summary

What Bandicoot does:

- Expression templates are used to collect the AST of an expression as a type at compile time.
- This AST is used at *compile time* to generate definitions of each of the macros above.
- At program runtime, the macros and skeleton kernel are combined and given to the GPU driver.
  (that is, if the kernel for the operation is not already cached)
- The linear algebra operation then runs (fast!).

What we get:

- The ‘usual’ speedups from expression template optimization.
- Kernels are automatically generated for arbitrary expressions.
- Kernel generation work is shifted into compile time instead of runtime.
- The compiled program contains only the (generated) kernels it needs.

Timing some operations

1. **Basic operations and submatrices:**

   `add2`: `Z = X + Y;`

   `add4`: `Z = A + B + C + D;`

   `chain`: `Z = A * B * C * D;` with decreasing sizes for each matrix.

   `addsub2`: same as `add2`, but using submatrices of `A` and `B`.

   `addsub4`: same as `add4`, but using submatrices of `A`, `B`, `C`, and `D`.

2. **Compound expressions:**

   `expr1`: `Z = 2 * (X.t() + Y) + 2 * (X + Y.t());`

   `expr2`: `Z = a * A + (B + C).t() + log(pow(D, 2));`

   `expr3`: `Z = 1 / (X % conv_to<fmat>::from(Y) + log(log(X + 2) % W));`

   `diagsum`: `(X.diag(-1) + X.diag(1)) % (Y.diag(-1) + Y.diag(1))`

3. **Neural network activations:**

   `relu`: `Z = X % (X > 0);`

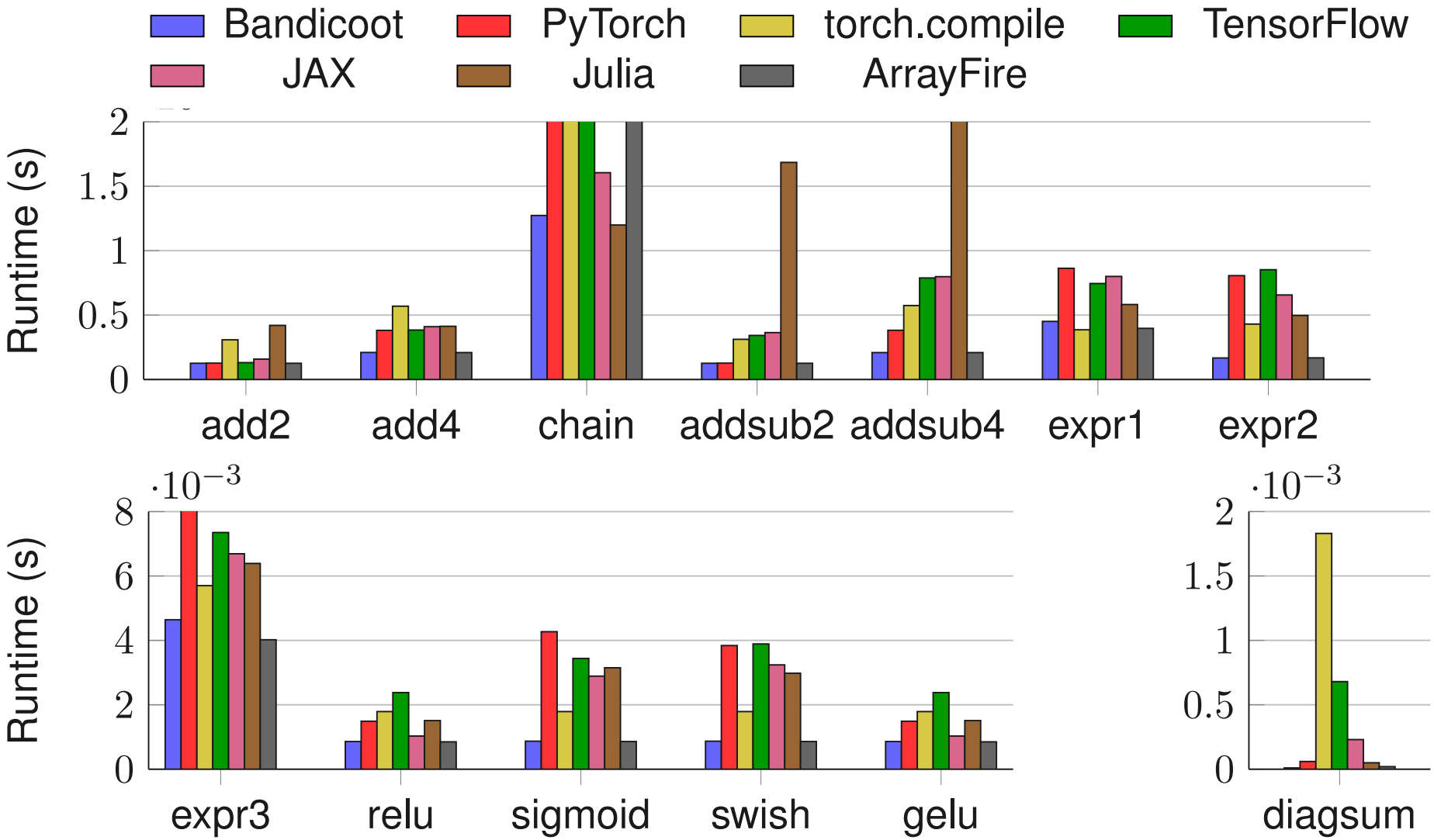
   `sigmoid`: `Z = 1 / (1 + exp(-X));`

   `swish`: `Z = X / (1 + exp(-beta * X));`

   `gelu`:
   ```
   Z = (X / 2) % (1 + tanh(sqrt(2 / pi) * (X + alpha *
       pow(X, 3))));
   ```

Timing results

CUDA backend, RTX4080, matrix size 10k x 10k, 50 trials.



Matrix chain addition bandwidth

How big can this expression get before we stop saturating GPU memory bandwidth?

```
z = x1 + x2 + x3 + x4 + ...
```

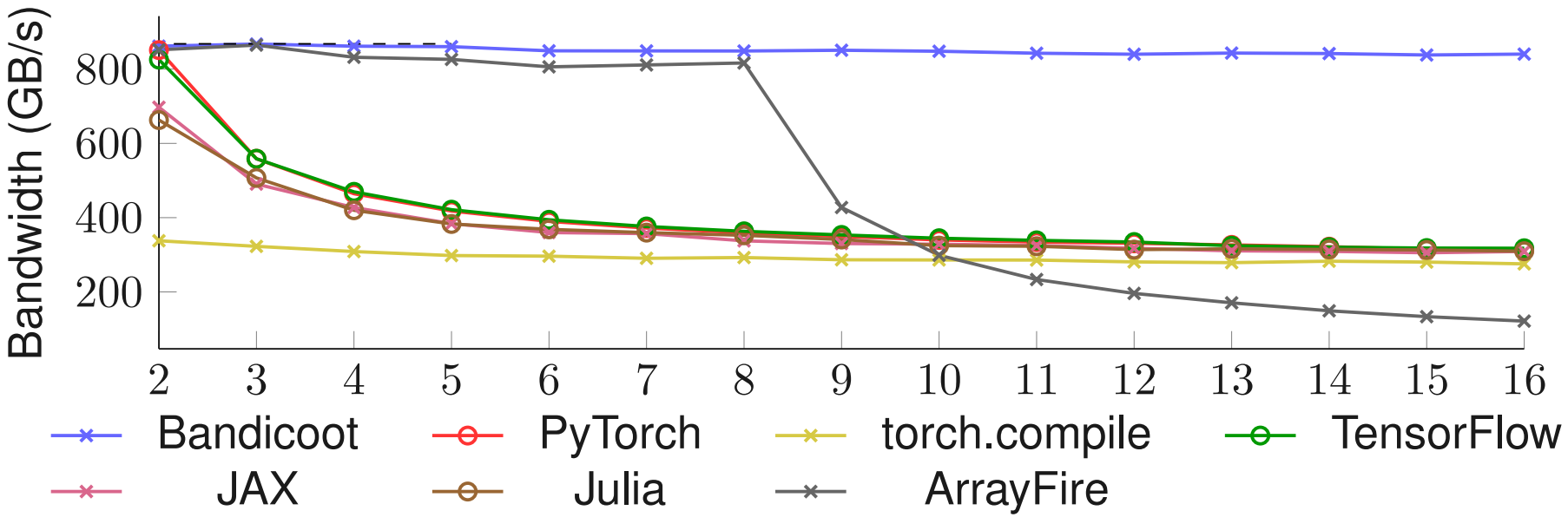
(RTX4090 peak bandwidth: 868 GB/s; 10k x 10k matrices again; 50 trials)

Matrix chain addition bandwidth

How big can this expression get before we stop saturating GPU memory bandwidth?

```
z = x1 + x2 + x3 + x4 + ...
```

(RTX4090 peak bandwidth: 868 GB/s; 10k x 10k matrices again; 50 trials)



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/
```

Short-term roadmap:

- More backends: Vulkan, Metal, HIP/ROCm
- Better support for decompositions and solvers
- Complex number support (currently only partial)
- Very-low-precision support (FP8/FP6/FP4)

How do you concatenate strings at compile time?

Warning: the path to insanity lies ahead.

First, define a templated utility struct to represent a compile-time character array. (You could use `std::array<char, N>`; we don't.)

```
template<size_t N>
struct char_array
{
  char data[N];

  constexpr const char& operator[](size_t i) const noexcept
  {
    return data[i];
  }

  ...
};
```

How do you concatenate strings at compile time?

Next, define some structs that can be used as compile-time strings.

```
struct str1
{
  // not counting null terminator
  static inline constexpr size_t len() { return 8; }
  static inline constexpr char_array<9> str()
  {
    return char_array<9>({ "string 1" });
  }
};

struct str2
{
  static inline constexpr size_t len() { return 9; }
  static inline constexpr char_array<10> str()
  {
    return char_array<10>({ " is great" });
  }
};
```

How do you concatenate strings at compile time?

Now a utility template struct to concatenate `char_array`s.

```
// base case, only called when Ts is empty (see the next slides)
template<typename... Ts>
struct concat_str
{
  static inline constexpr size_t len() { return 0; }
  static inline constexpr char_array<1> str()
  {
    return char_array<1>{ '\0' };
  }
};
```

How do you concatenate strings at compile time?

Now a utility template struct to concatenate `char_array`s.

```
// case for when only one string is given
template<typename T1>
struct concat_str
{
  static inline constexpr size_t len() { return T1::len(); }
  static inline constexpr char_array<len() + 1> str()
  {
    return T1::str();
  }
};
```

How do you concatenate strings at compile time?

```
template<typename T1, typename... Ts> // recursive case
struct concat_str
{
  static inline constexpr size_t len()
  {
    return T1::len() + concat_str<Ts...>::len();
  }
  static inline constexpr char_array<len() + 1> str()
  {
    char_array<len() + 1> r {};
    constexpr const size_t l1 = arg_len<T1>::len();
    constexpr const size_t l2 = arg_len<T2>::len();
    constexpr const auto s1 = T1::str();
    constexpr const auto s2 = T2::str();

    size_t p = 0;
    for (size_t i = 0; i < l1; ++i, ++p) r[p] = s1[i];
    for (size_t i = 0; i < l2; ++i, ++p) r[p] = s2[i];
    r[len()] = '\0';
    return r;
  }
};
```