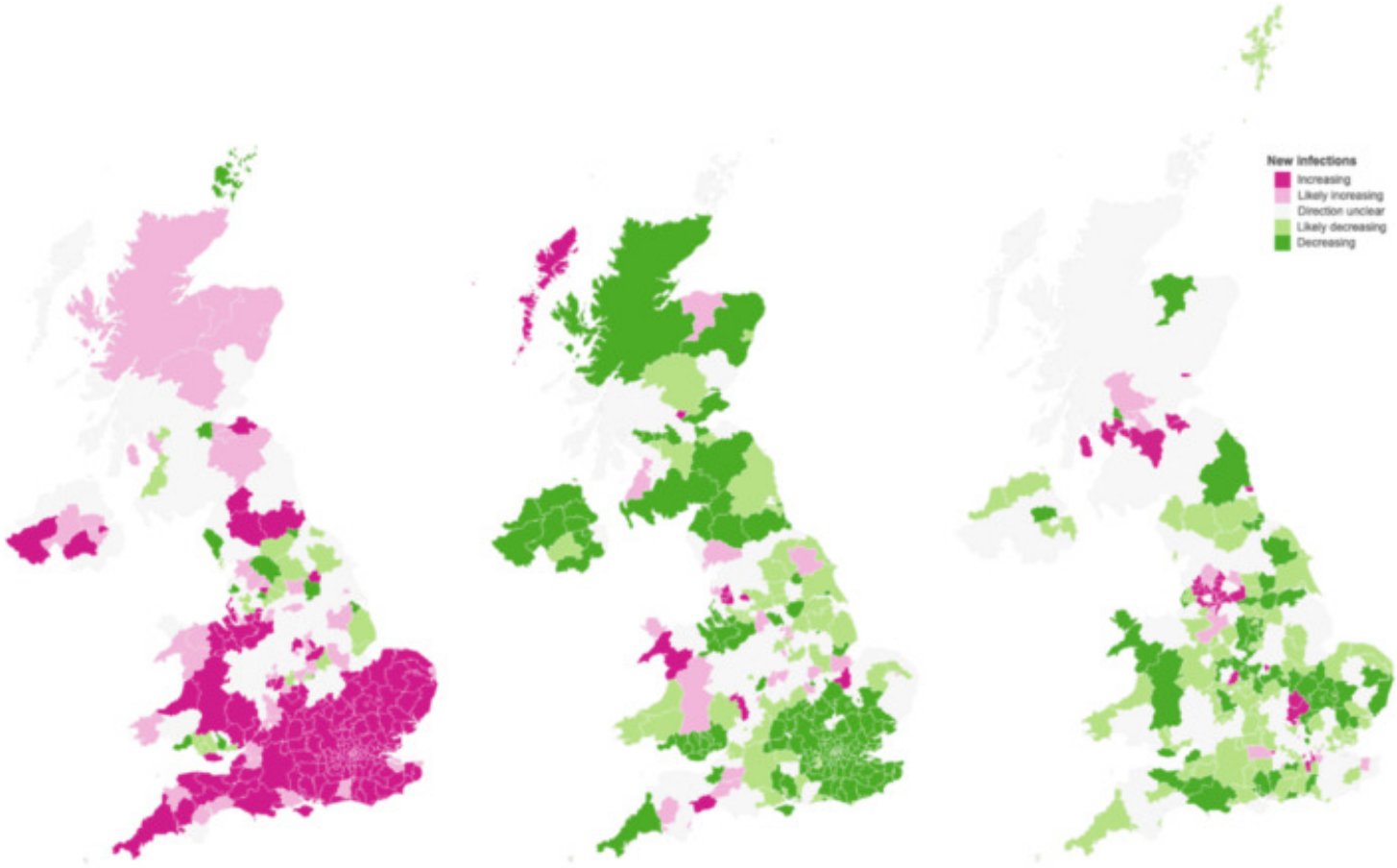
From Bayesian Inference to LLMs

Modern C++ Optimizations for Reverse-Mode Automatic Differentiation

STEVE BRONDER

Estimating COVID Infection Rates For Policy



(a) 25 Dec 2020 (b) 21 Jan 2021 (c) 30 May 2021

Figure: Probability of epidemic growth by local area

What is Automatic Differentiation?

Computational method to calculate gradients of functions in a program by systematically applying the chain rule

Why use Automatic Differentiation

Ex: Newton's root finding method

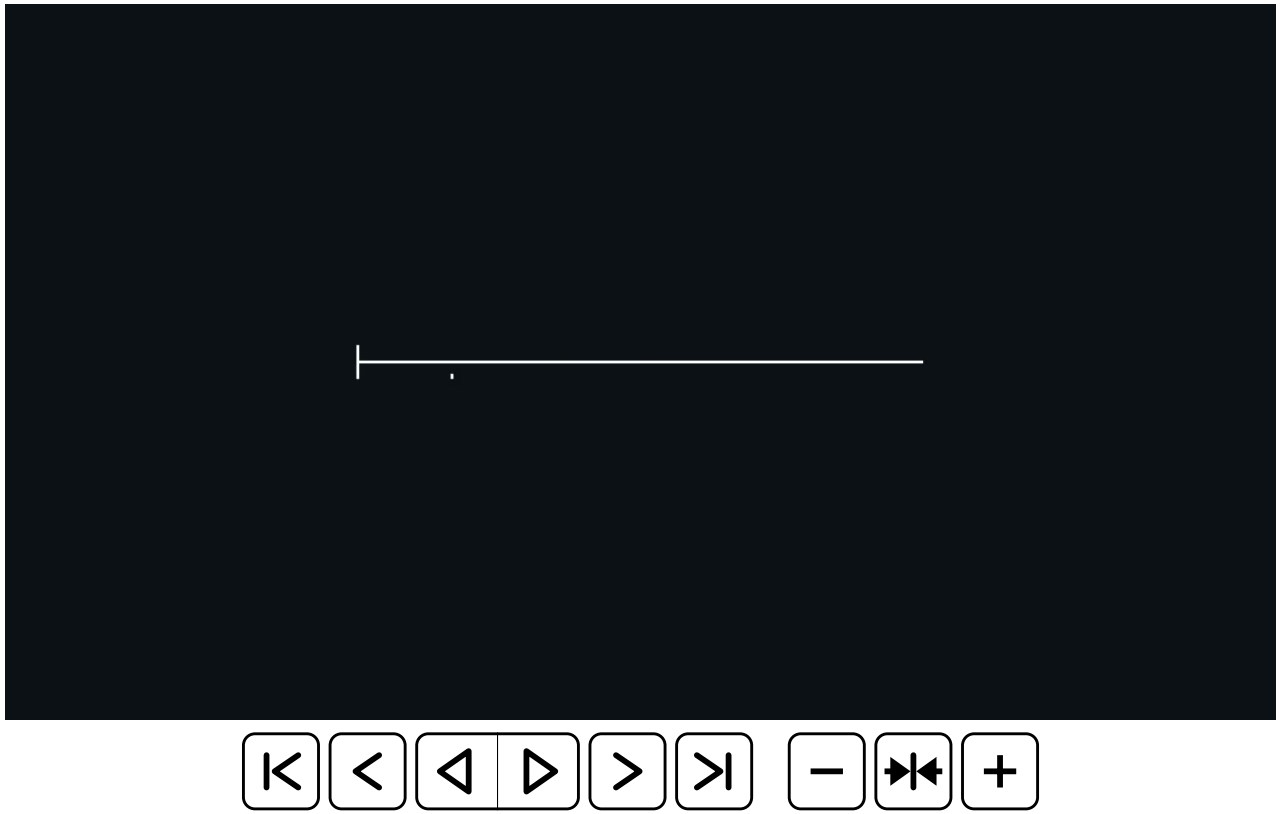
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

find $f(x) = y$ where $y = 0$

Why use Automatic Differentiation?

$$f(x) = x^3 + x^2 + x \tag{1}$$

$$f'(x) = 3x^2 + 2x + 1 \tag{2}$$



Why use Automatic Differentiation?

- Hamiltonian Monte Carlo:

$$\frac{dp}{dt} = \nabla_\theta \log p(\theta|y)$$

- BFGS:

$$s_k = -H_k \nabla_\theta f(\theta_k)$$

- Stochastic Gradient Descent:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x_t)$$

Why use Automatic Differentiation?

Choices:

- Write by hand
- finite difference,
- symbolic differentiation
- spectral differentiation
- automatic differentiation

Why use Automatic Differentiation?

$$\underbrace{p(\boldsymbol{\theta} \mid \mathbf{y})}_{\textbf{posterior}} \propto \prod_{i=1}^{N} \left\{ \sum_{\mathbf{z}_i \in \{1,2,3\}^{T_i}} \left[\underbrace{\pi_{z_{i,1}} \prod_{t=2}^{T_i} \Pi_{z_{i,t-1},\, z_{i,t}}}_{\text{3-state HMM prior}} \prod_{t=1}^{T_i} \underbrace{\mathcal{N}\left(y_{i,t} \,\middle|\, \eta_{i,t},\ \sigma^2_{z_{i,t}}\right)}_{\text{state-dependent emission}} \right] \right\}$$

where $\eta_{i,t} = \underbrace{\mathbf{x}_{i,t}^\top \boldsymbol{\beta}}_{\text{fixed}} + \underbrace{\mathbf{z}_{i,t}^{(G)\top} \mathbf{b}_{g[i]} + \mathbf{z}_{i,t}^{(C)\top} \mathbf{c}_{c[i]} + u_{g[i]}}_{\text{crossed random effects}} + \underbrace{f(t_{i,t})}_{\text{GP}} + \underbrace{\mu_{z_{i,t}} + \mathbf{r}_{z_{i,t}}^\top \mathbf{w}_i}_{\text{state-specific offset + slope}}, \qquad z_{i,t} \in \{1, 2, 3\}.$

$$p(\mathbf{f} \mid \boldsymbol{\psi}) = (2\pi)^{-T/2} \, |\mathbf{K}|^{-1/2} \, \exp\Big(-\tfrac{1}{2} \mathbf{f}^\top \mathbf{K}^{-1} \mathbf{f} \Big),$$

$$\mathbf{K} = \sigma_f^2 \Big(\mathbf{K}_{\mathsf{LP}}(\ell, p, \lambda) \ + \ \rho \, \mathbf{K}_{\mathsf{SE}}(\tilde{\ell}) \Big) + \sigma_n^2 \mathbf{I}, \quad \big[\mathbf{K}_{\mathsf{LP}}\big]_{tt'} = \exp\left(-\frac{(t-t')^2}{2\ell^2} - \frac{2\sin^2\left(\pi |t-t'|/p\right)}{\lambda^2} \right),$$

$$\big[\mathbf{K}_{\mathsf{SE}}\big]_{tt'} = \exp\left(-\frac{(t-t')^2}{2\tilde{\ell}^2} \right), \qquad \mathbf{K} = \mathbf{L}_K \mathbf{L}_K^\top \ \Rightarrow \ \log|\mathbf{K}| = 2 \sum_{j=1}^{T} \log\left((\mathbf{L}_K)_{jj} \right).$$

Hierarchical mixed effects (non-centered, LKJ prior):

$$\mathbf{b}_g = \left(\mathbf{I}_{p_G} \otimes \operatorname{diag}(\boldsymbol{\tau}_b) \, \mathbf{L}_R \right) \tilde{\mathbf{b}}_g, \quad \tilde{\mathbf{b}}_g \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad \mathbf{c}_c = \operatorname{diag}(\boldsymbol{\tau}_c) \, \tilde{\mathbf{c}}_c, \quad \tilde{\mathbf{c}}_c \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

$$\mathrm{LKJ}_{p_G}(\eta) \text{ prior on } \mathbf{R}, \quad \mathbf{L}_R \mathbf{L}_R^\top = \mathbf{R}, \quad \boldsymbol{\tau}_b \sim \prod_{j=1}^{p_G} \text{Half-}t_{\nu_b}(0, s_b), \quad \boldsymbol{\tau}_c \sim \prod_{j=1}^{p_C} \text{Half-}t_{\nu_c}(0, s_c),$$

$$u_g \sim \mathcal{N}(0, \sigma_u^2).$$

Why use Automatic Differentiation?

- I do not wish to write those gradients by hand

Why use Automatic Differentiation?

- I do not wish to write those gradients by hand
- Faster than finite difference, more flexible than symbolic differentiation

Why use Automatic Differentiation?

- I do not wish to write those gradients by hand
- Faster than finite difference, more flexible than symbolic differentiation
- Allows for unknown length while and for loops

Why use Automatic Differentiation?

- I do not wish to write those gradients by hand
- Faster than finite difference, more flexible than symbolic differentiation
- Allows for unknown length while and for loops
- Accurate to floating point precision

Why use Automatic Differentiation?

- I do not wish to write those gradients by hand
- Faster than finite difference, more flexible than symbolic differentiation
- Allows for unknown length while and for loops
- Accurate to floating point precision
- Reverse Mode AD can compute partials derivatives of inputs at the same time

Implementation Matters!

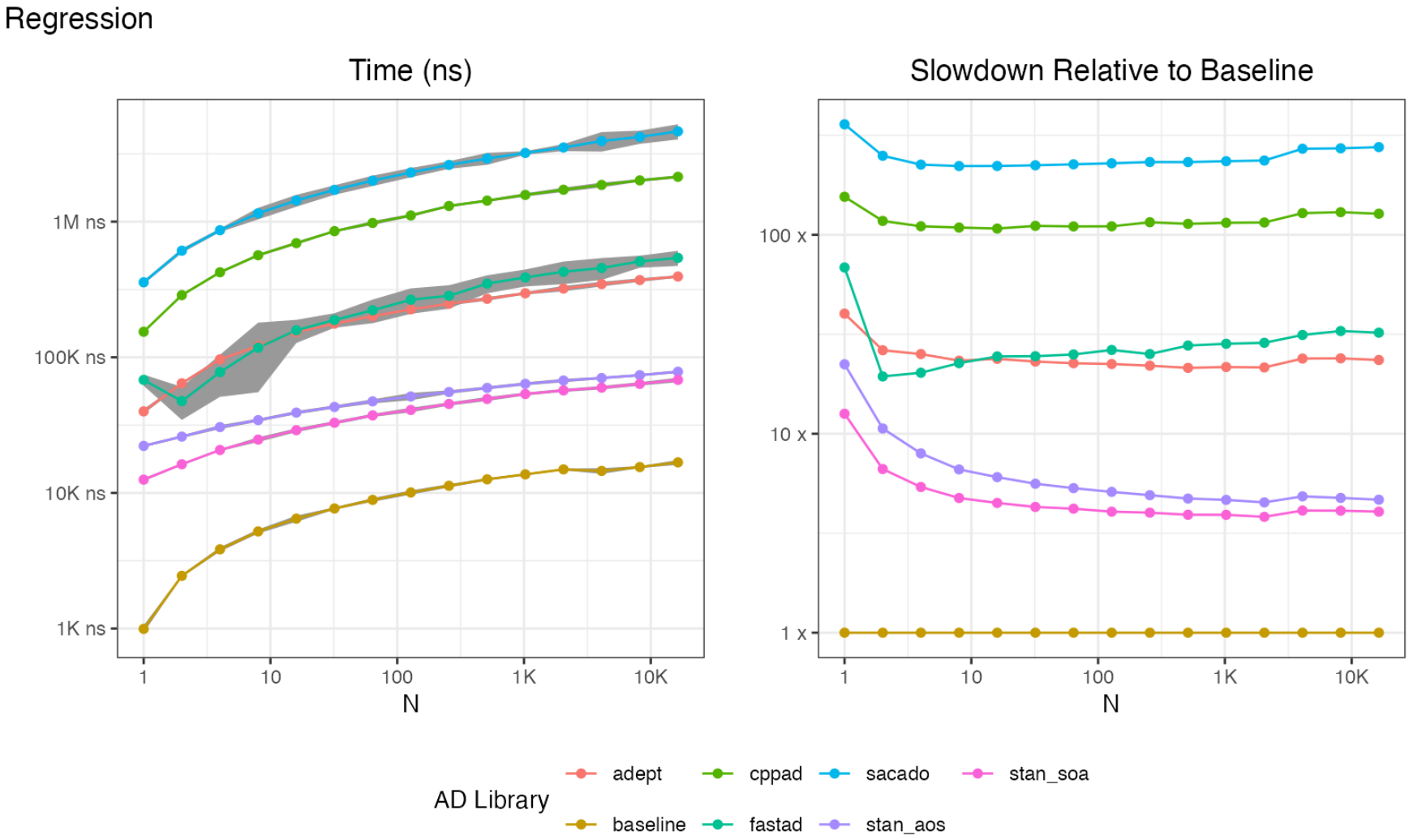


Figure: Benchmark for f' given $f(y) = N(y|X\theta, \sigma)$

Goal of this talk

- How Reverse Mode Automatic Differentiation works
- Performance of high throughput memory intensive programs
- Show how modern C++ has led to cleaner and more efficient AD

What is Automatic Differentiation?

- AD computes gradients of a program by applying the chain rule to its subexpressions.

$$f(x, y) = \log(x \cdot y)$$

$$f_1(x, y) = x \cdot y \qquad \frac{\partial f_1}{\partial x} = y \qquad \frac{\partial f_1}{\partial y} = x$$

$$f_2(u) = \log(u) \qquad \frac{\partial f_2}{\partial u} = \frac{1}{u}$$

$$f(x, y) = f_2\big(f_1(x, y)\big)$$

What is Automatic Differentiation?

- AD computes gradients of a program by applying the chain rule to its subexpressions.

$$f(x, y) = \log(x \cdot y)$$

$$f_1(x, y) = x \cdot y \qquad \frac{\partial f_1}{\partial x} = y \qquad \frac{\partial f_1}{\partial y} = x$$

$$f_2(u) = \log(u) \qquad \frac{\partial f_2}{\partial u} = \frac{1}{u}$$

$$f(x, y) = f_2\big(f_1(x, y)\big)$$

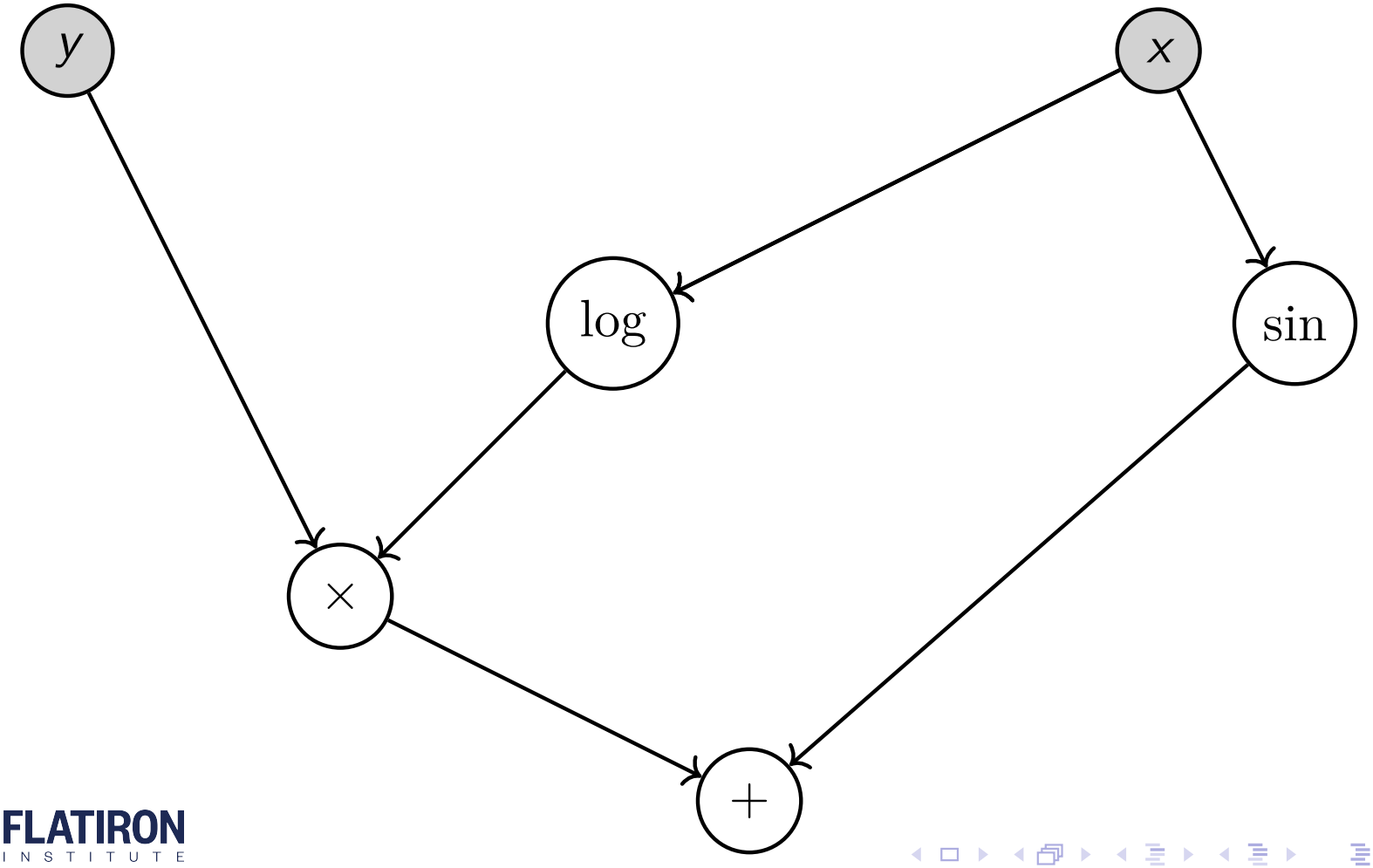
$$\frac{\partial f}{\partial x} = \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x} = \frac{1}{f_1(x, y)} \cdot y = \frac{y}{x \cdot y} = \frac{1}{x},$$

$$\frac{\partial f}{\partial y} = \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial y} = \frac{1}{f_1(x, y)} \cdot x = \frac{x}{x \cdot y} = \frac{1}{y}.$$

Data Type: Expression Graph

Goal: Calculate the full gradient by accumulating partial gradients (adjoints) through the a graph of subexpressions.

$$z = \log(x) \cdot y + \sin(x)$$



What is Automatic Differentiation

For Reverse Mode AD, each node performs two functions.

- Forward Pass:

$$v_2 = f(v_1, v_0)$$

- Reverse Pass: Given v_2's adjoint (partial gradient) $\overline{v_2}$
  Calculate the local adjoint-Jacobian update for v_1 and v_0.

$$chain(\overline{v_2}, v_1, v_0) = \left\{ \overline{v_1} \mathrel{+}= \frac{\partial v_2}{\partial v_1} \overline{v_2}, \overline{v_0} \mathrel{+}= \frac{\partial v_2}{\partial v_0} \overline{v_2} \right\}$$

What Do AD Libraries Care About?

- Flexibility:

  Debugging, exceptions, conditional loops, matrix subset assignment

- : Efficiency:

  Efficiently using a single CPU/GPU

- Scaling

  Efficiently using clusters with multi-gpu/cpu nodes

How do we keep track of our reverse pass?

- Source code transformation

  Unroll all forward passes and reverse passes into one function

  Good: Fast

  Bad: Hard to implement, very restrictive

- Operator Overloading

  Nodes in the expression graph are objects which store a forward and reverse pass function

  Good: Easier to implement, more flexible

  Bad: Less optimization opportunities

Newer AD packages use a combination of both

How do we keep track of our reverse pass?

Static (Fast) vs. Dynamic (Flexible) graphs

- Known expression graph size at compile time? (Static)
- Reassignment of variables (Dynamic easy, Static hard)
- How much time do I have? (Dynamic)

Operator Overloading Approach

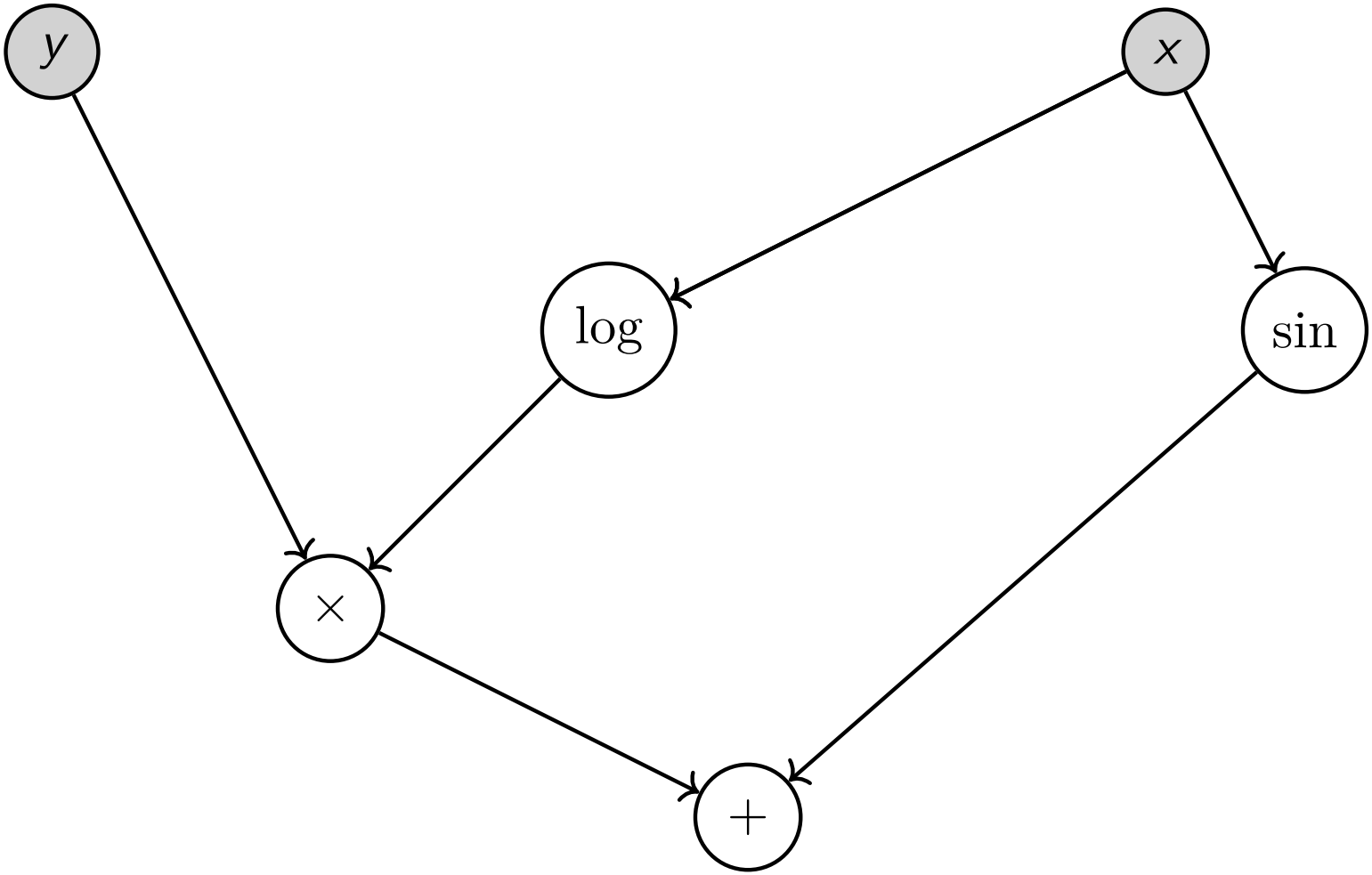
The operator overloading approach usually involves:

- Tracking the expression graph of reverse pass function calls
- A pair scalar type to hold the value and adjoint

Allows conditional loops and reassignment of values in matrices

Make A Tape

$$z = \log(x) \cdot y + \sin(x)$$



Tape of Expression Graph

$$f(x, y) = \log(x)y + \sin(x)$$

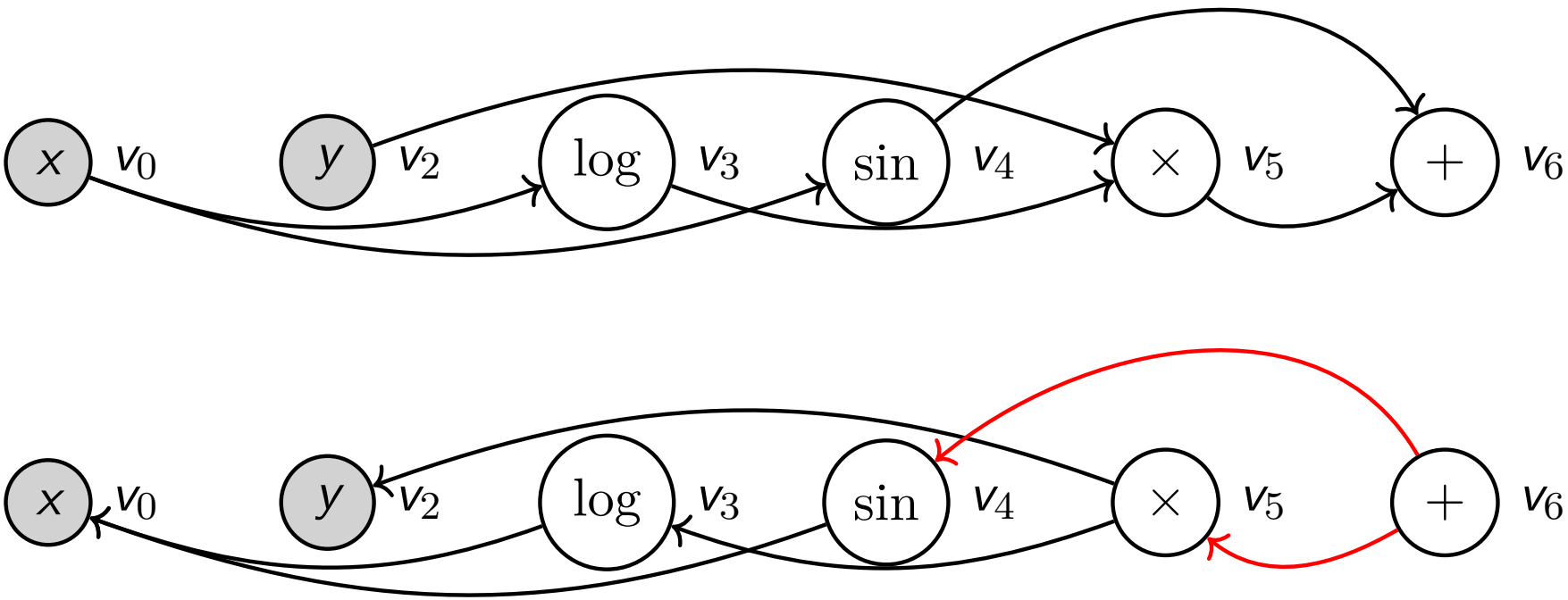


Figure: Topological sort of expression graph

Monotonic Buffer

```
struct Tape {
std::array<std::byte, (1 << 16)> buf;
std::monotonic_buffer_resource mbr{buf.data(),
↪  buf.size()};
std::polymorphic_allocator<std::byte> pa{&mbr};
std::vector<var_impl*> tape;
template <typename T, typename Types>
auto* new_node(Types&&... args) {
  auto* ret = pa.template new_object<T>(args...);
  g_tape.tape.emplace_back(ret);
  return ret;
}
void clear() {
  tape.clear();
  mbr.release();
}
};
static Tape g_tape{};
```

Monotonic Buffer

```
struct var_impl {
  double val;
  double adj;
  virtual void chain() {};
}; // 24 bytes
struct var {
  var_impl* vi_;
  var(double val) :
    vi_(g_tape.pa.template
      ↪  new_object<var_impl>(val)) {}
}; // 8 bytes
```

Monotonic Buffer

```
struct mul_vv : public var_impl {
  var op1_;
  var op2_;
  void chain() {
    op1_.adj() += this->adjoint_ * op2_.val();
    op2_.adj() += this->adjoint_ * op1_.val();
  }
}; // 40 bytes
operator*(var op1, var op2) {
  return var{g_tape.template new_node<mul_vv>(
    op1.val() * op2.val(), op1, op2)};
}
```

Monotonic Buffer

```
void compute_grads(var x, var y) {
  var z = -10;
  while (z.val() < 20) {
    z += log(x * y);
  }
  grad(z);
}
// Later in program
for (int i = 0; i < 1e10; ++i) {
  var x = compute_x(...);
  var y = compute_y(...);
  compute_grads(x, y);
  do_something_with_grads(x.adj(), y.adj());
  g_tape.clear();
}
```

So Many Operator Classes

```
struct mul_vv;
struct mul_vd;
struct mul_dv;
struct add_vv;
struct add_dv;
struct add_vd;
struct subtract_vv;
struct subtract_dv;
struct subtract_vd;
struct divide_vv;
struct divide_dv;
struct divide_vd;
```

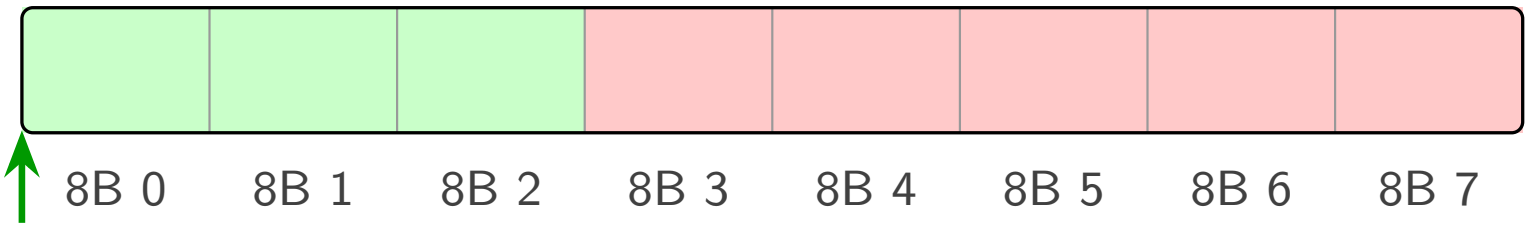
Reduce Boilerplate

```cpp
template <Functor F>
struct callback_var_impl : public var_impl {
  F rev_functor_;
  template <std::floating_point S>
  explicit callback_var_impl(
    S&& value, F&& rev_functor)
    : var_impl(value),
      rev_functor_(rev_functor) {}
  void chain() final { rev_functor_(*this); }
}; // 24 + 8B * N
// Helper for callback_var_impl
template <std::floating_point FwdVal, typename F>
auto lambda_var(FwdVal val, F&& rev_functor) {
  return var(
    g_tape.template new_node<callback_var_impl<F>>(
      val,
      std::forward<F>(rev_functor)));
}
```

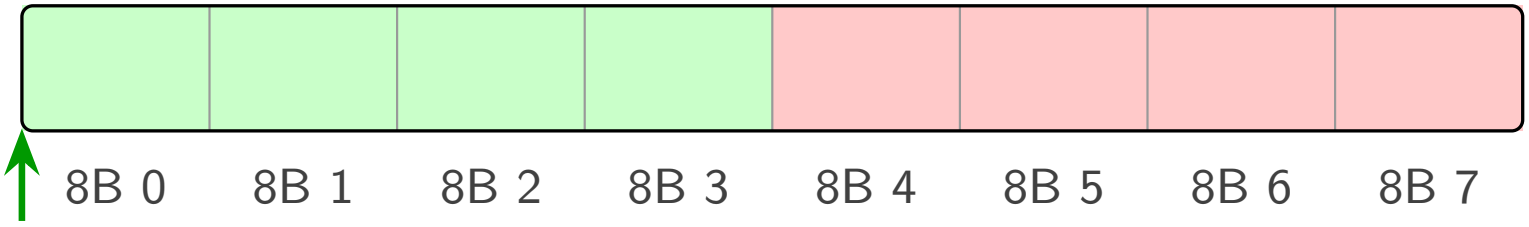
Reduce Boilerplate

```
template <typename T1, typename T2>
requires any_var<T1, T2>
inline auto operator*(T1 op1, T2 op2) {
  return lambda_var(value(op1) * value(op2),
    [op1, op2](auto&& ret) mutable {
      if constexpr (is_var_v<T1>) {
        adjoint(op1) += adjoint(ret) * value(op2);
      }
      if constexpr (is_var_v<T2>) {
        adjoint(op2) += adjoint(ret) * value(op1);
      }
    });
}
```

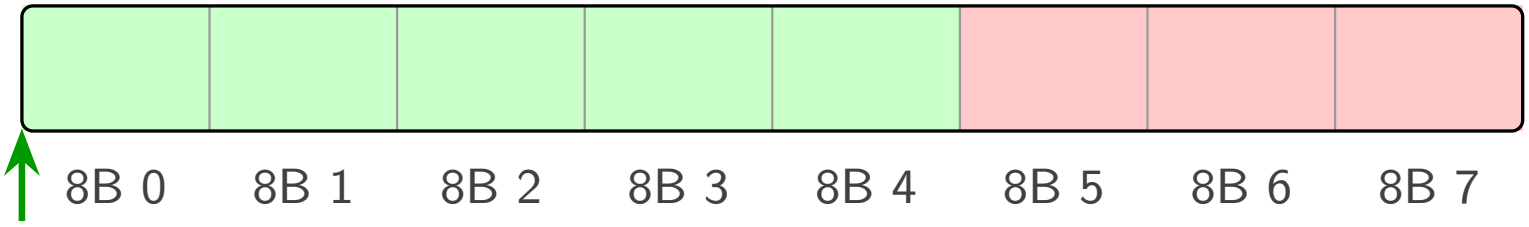
Poor Cache Use



var_impl{dbl, dbl, vptr}



unary_var{dbl, dbl, vptr, var_impl*}



binary_var{dbl, dbl, vptr, var_impl*, var_impl*}

Source Code Transform Ex:

```
double z = log(x * y);
```

Break it down

```
double v0 = x;
double v1 = y;
double v2 = x * y;
double v3 = log(v2)
double bar_v3 = 1;
double bar_v2 = bar_v3 * 1/v2;
double bar_v1 = bar_v2 * v0;
double bar_v0 = bar_v2 * v1;
```

Source Code Transform Ex:

Code like the following very hard / impossible in source code transform

```
while(error < tolerance) {
  //  ...
}
```

Source Code Transform Ex:

```
struct var {
    double values_;
    double adjoints_;
    var(double x) : values_(x), adjoints_(0) {}
};
```

Source Code Transform Ex:

```
template <typename F, typename... Exprs>
struct expr {
   var ret_;
   std::tuple<deduce_ownership_t<Exprs>...> exprs_;
   std::decay_t<F> f_;
   template <typename FF, typename... Args>
   expr(double x, FF&& f, Args&&... args) :
     ret_(x), f_(std::forward<F>(f)),
     exprs_(std::forward<Args>(args)...) {}
};
```

Source Code Transform Ex:

```
template <typename T1, typename T2>
requires any_var_or_expr<T1, T2>
inline auto operator*(T1&& lhs, T2&& rhs) {
  return make_expr(value(lhs) * value(rhs),
  [](auto&& ret, auto&& lhs, auto&& rhs) {
    if constexpr (!std::is_arithmetic_v<T1>) {
      adjoint(lhs) += adjoint(ret) * value(rhs);
    }
    if constexpr (!std::is_arithmetic_v<T2>) {
      adjoint(rhs) += adjoint(ret) * value(lhs);
    }
  }, std::forward<T1>(lhs), std::forward<T2>(rhs));
}
```

Source Code Transform Ex:

```
auto z = x * log(y) + log(x * y) * y;
expr<Lambda<Plus>,
  expr<Lambda<Mult>, var, expr<Lambda<Log>, var>>,
  expr<Lambda<Mult>,
   expr<Lambda<Log>, expr<Lambda<Mult>, var, var>>,
   var>>
```

Source Code Transform Ex:

```
template <typename Expr>
inline void grad(Expr&& z) {
  adjoint(z) = 1.0;
  auto nodes = collect_bfs(z);
  eval_breadthwise(nodes);
}
```

Example Godbolt

Comparison

Table: $f(x, y) = x\log(y) + \log(xy)y$;

Method	CPU Time	% Improvement
Shared Ptr	508ns	1.0
MonoBuff	121ns	3.9x
Lambda	112ns	4.2x
Source Code Transform	26.5ns	19x
Baseline	2.82ns	180x

Operator Overloading Approach: Matrices

```
Matrix<var> B(M, M);
Matrix<var> X(M, M);
Matrix<var> Z = X * B.transpose();
```

Operator Overloading Approach: Matrices

```
template <typename MatrixType>
struct arena_matrix :
  public Eigen::Map<MatrixType> {
  using Base = Eigen::Map<MatrixType>
  template <typename T>
  arena_matrix(T&& mat) :
   Base(copy_to_arena(mat.data(), mat.size()),
     mat.rows(), mat.cols()) {}
};
```

Operator Overloading Approach: Matrices

```
// Array of Structs
struct Matrix<var> {
  var* data_;
};
// Struct of Arrays
struct var_impl<Matrix<double>> {
  arena_matrix<double> value_;
  arena_matrix<double> adjoint_;
  virtual void chain() {}
};
```

Operator Overloading Approach: Matrices

```
// Array of Structs
struct Matrix<var> {
  var* data_;
};
```

- Array of Structs:
  - Simple, most algorithms Just Work™
  - Adds a lot to expression graph
  - turns off SIMD

Operator Overloading Approach: Matrices

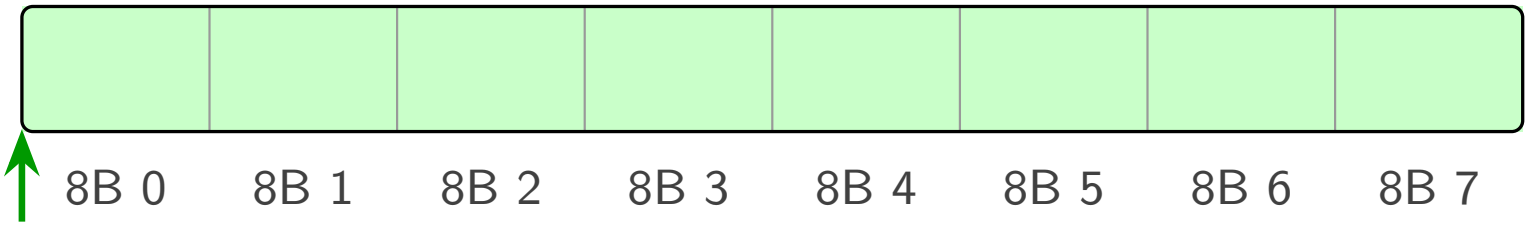
```
// Struct of Arrays
struct var_impl<Matrix<double>> {
  arena_matrix<double> value_;
  arena_matrix<double> adjoint_;
  virtual void chain() {}
};
```

- Struct of Arrays:
  - Hard, everything written out manually
  - Collapses matrix expressions in tree
  - SIMD can be used on values and adjoints

Matrix Multiplication Example

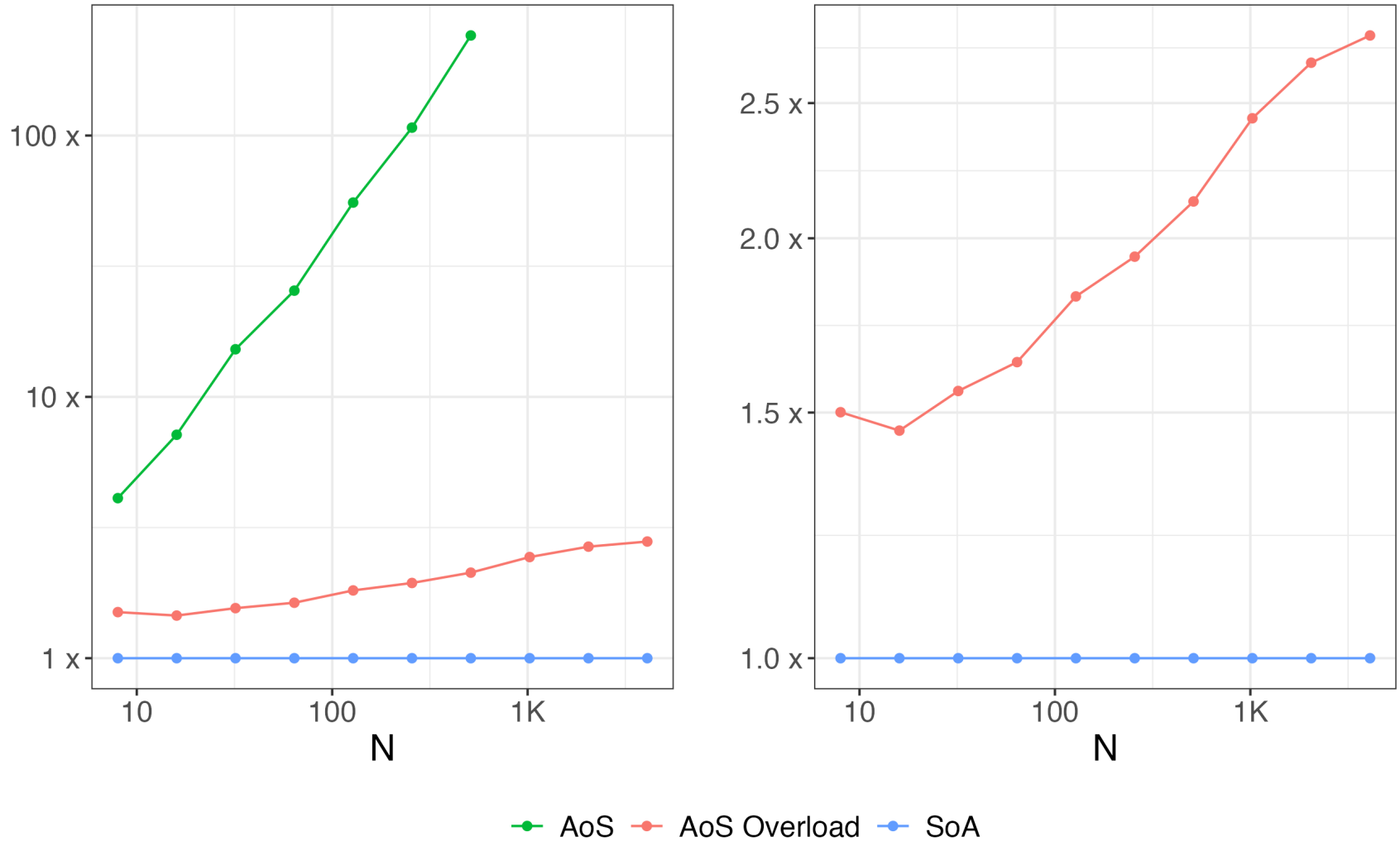
```
template <typename T1, typename T2>
requires any_var_matrix<T1, T2>
inline auto operator*(T1&& op1, T2&& op2) {
  return lambda_var(value(op1) * value(op2),
    [op1, op2](auto&& ret) mutable {
      if constexpr (is_var_matrix_v<T1>) {
        adjoint(op1) += adjoint(ret) *
          ↪  value(op2).transpose();
      }
      if constexpr (is_var_matrix_v<T2>) {
        adjoint(op2) += value(op1) * adjoint(ret);
      }
    });
}
```

Operator Overloading Approach: Matrices



mat_mul_vv{dbl*, dbl*, long int, long int, vptr, lambda[var<Mat>, var<Mat>]}

Matrix Multiplication Benchmark

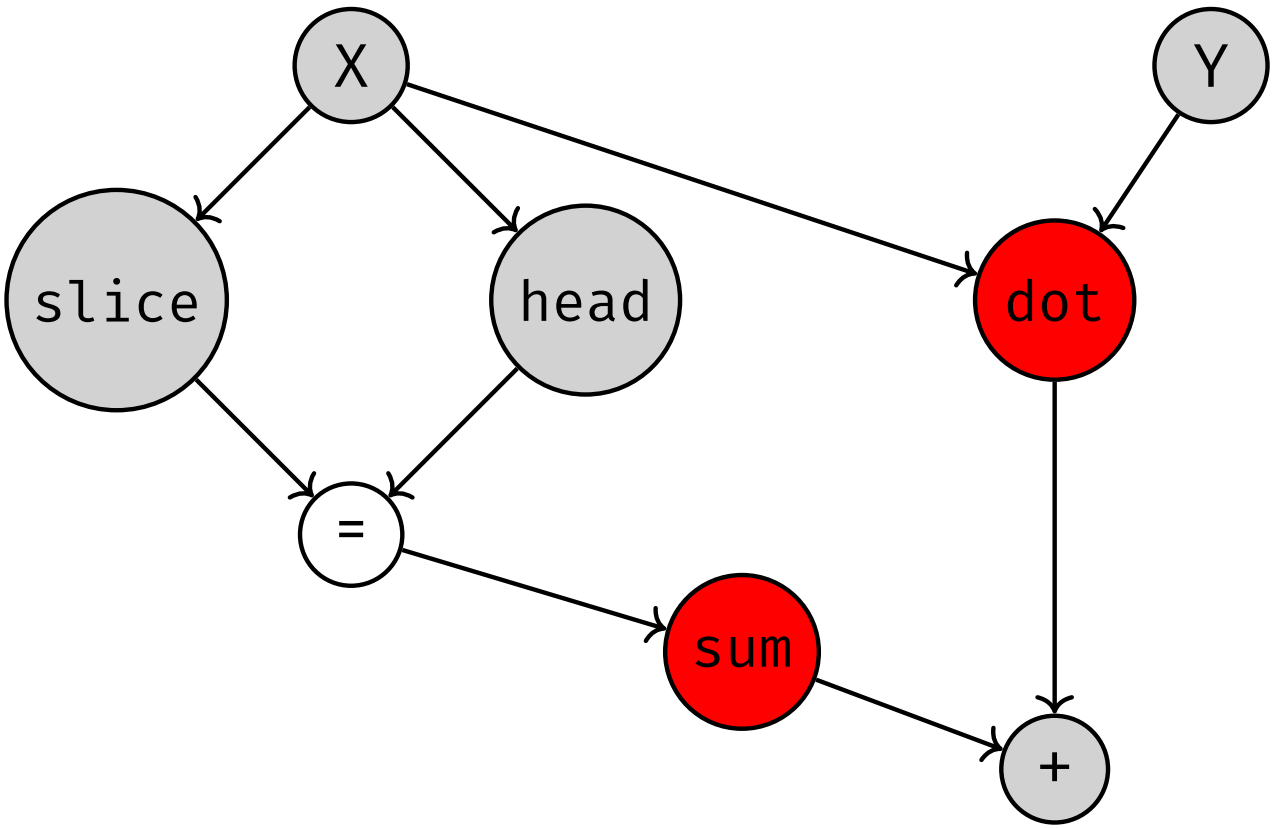


Subset Assignment

```
var<Vector<double>> y{{0, 1, 2, 3}};
var<Vector<double>> x{{0, 1, 2, 3}};
var prod = y.dot(x);
x.slice(1, 3) = x.head(3);
auto z = prod + sum(x);
```

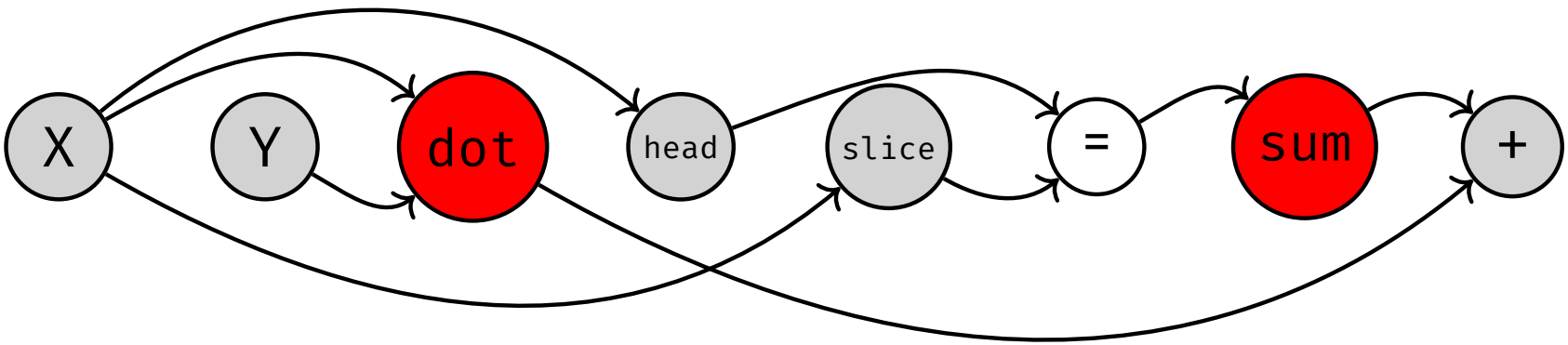
Subset Assignment

```
var<Vector<double>> y{{0, 1, 2, 3}};
var<Vector<double>> x{{0, 1, 2, 3}};
var prod = y.dot(x);
x.slice(1, 3) = x.head(3);
auto z = prod + sum(x);
```



Subset Assignment

```
var<Vector<double>> y{{0, 1, 2, 3}};
var<Vector<double>> x{{0, 1, 2, 3}};
var prod = y.dot(x);
x.slice(1, 3) = x.head(3);
auto z = prod + sum(x);
```



Subset Assignment Becomes Very Hard

```
var<Vector<double>> x{{0, 1, 2, 3}};
x.slice(1, 3) = x.head(3);
```

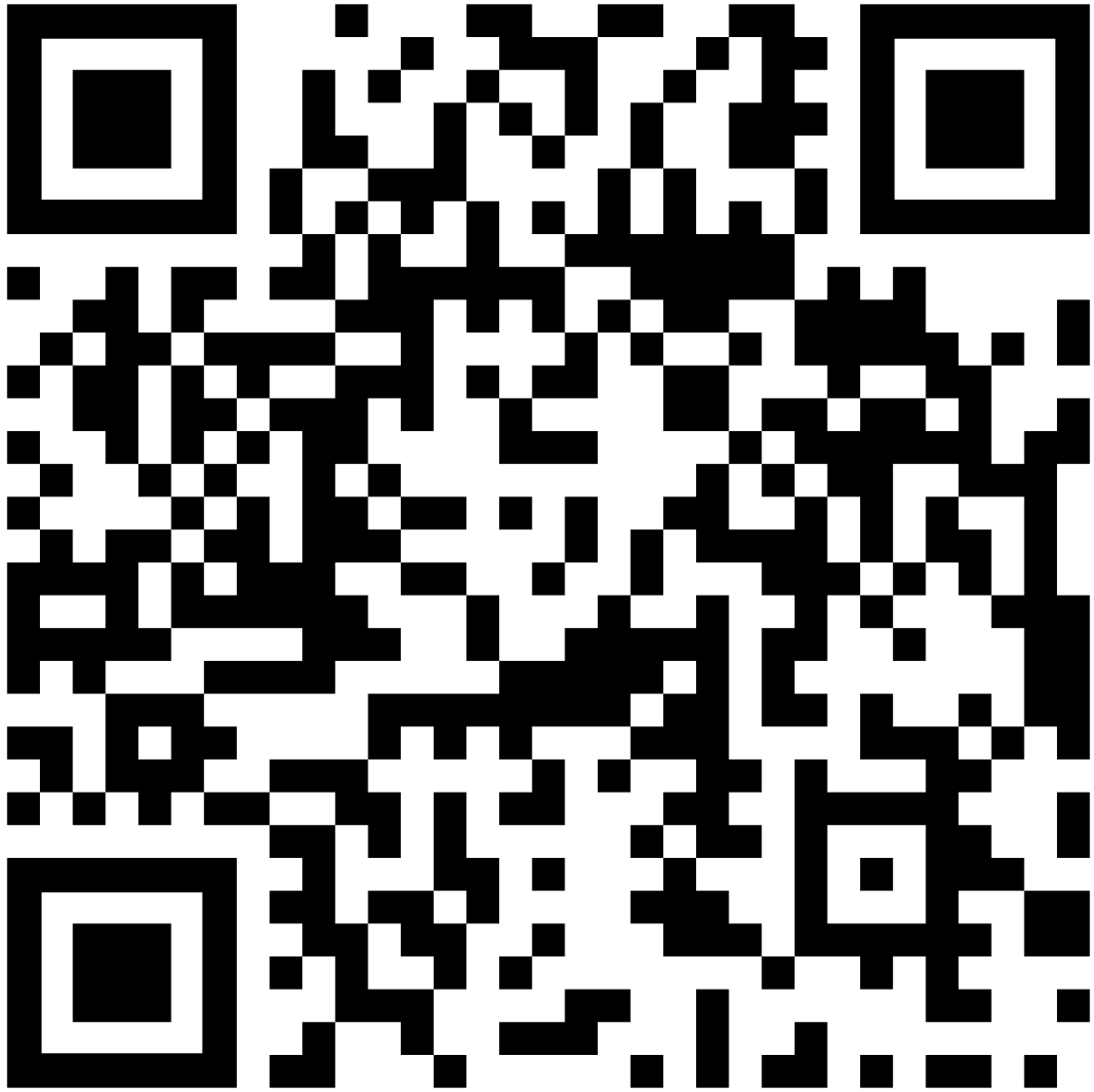
Iter	X
0	{0, 0, 2, 3}
1	{0, 0, 0, 3}
2	{0, 0, 0, 0}

Subset Assignment Becomes Very Hard

```
tempalte <typename T>
struct var {
template <typename S>
require AssignableExpression<T, S>
inline var<T>& operator=(const var<S>& other) {
  arena_matrix<T> prev_val(vi_->val_.rows(),
  ↪  vi_->val_.cols());
  prev_val.deep_copy(vi_->val_);
  vi_->val_.deep_copy(other.val());
  g_tape.callback(
  [this_vi = this->vi_, other_vi = other.vi_,
  ↪  prev_val]() mutable {
    this_vi->val_.deep_copy(prev_val);
    prev_val.deep_copy(this_vi->adj_);
    this_vi->adj_.setZero();
    other_vi->adj_ += prev_val;
  });
  return *this;
}
}
```

Thanks!

Repository for benchmarks and slides



FLATIRON INSTITUTE