Support usage of Thrust API in Clad

Author: Abdelrhman Elrawy
Mentors: Vassil Vassilev, Alexander Penev

Project Context

Clad: A source-transformation automatic differentiation (AD) library in Clang.

Thrust: NVIDIA's powerful GPU-parallel algorithms and data structures library.

The Challenge:

- This project aims to enhance Clad by adding support for NVIDIA's Thrust library.
- By enabling differentiation of Thrust's GPU-parallel algorithms, Clad users will gain the ability to automatically generate gradients for CUDA-accelerated code.
- This work will bridge the gap between high-performance GPU computing and AD, potentially accelerating gradient-based optimization tasks by orders of magnitude.

Project Progress Summary

16 Pull Requests Merged

- **Core Algorithms** (8 PRs):
    1. thrust::reduce - Parallel reductions with multiple binary operators
    2. thrust::inner_product - Dot products and inner products
    3. thrust::transform - Element-wise transformations
    4. thrust::transform_reduce - Fused transform and reduce operations
    5. thrust::copy - Memory operations with gradient tracking
    6. thrust::adjacent_difference - Compute differences between adjacent elements
    7. Reduction overloads - Additional operator support
    8. Reverse-forward mode for reduce

Project Progress Summary

- **Advanced Operations (4 PRs):**
  1. Scan operations - Inclusive/exclusive prefix sums (fundamental parallel primitive)
  2. thrust::sort_by_key - Sort key-value pairs with gradient preservation
  3. thrust::reduce_by_key - Segmented reductions for grouped data
  4. Segmented scans - Advanced partitioned prefix sum operations

- **Infrastructure (2 PR):**
  1. thrust::device_vector support
  2. Added Generic functor support for transform

- **Demonstrations (2 PRs):**
  1. Multiple Thrust-based demo applications (Linear Regression, Particle simulation)
  2. Bag-of-Words Logistic Regression - ML Demo

Core Algorithm Implementations

- Reduction Operations
  1. thrust::reduce
     - Parallel sum, max, min, product operations
     - Special handling for mathematical edge cases (zeros in multiplication)
     - Multiple binary operator support
  2. thrust::inner_product
     - Dot products with customizable operations
     - 4-argument and 6-argument versions
     - Essential for linear algebra on GPU

Core Algorithm Implementations

- Reduction Operations
    1. thrust::reduce

```
void reduce_pullback(Iterator first, Iterator last, T init, BinaryOp op,
                     T d_output, Iterator* d_first, Iterator* d_last, T* d_init,
                     BinaryOp* d_op) {
  size_t n = ::thrust::distance(first, last);

  auto d_first_const_ptr = ::thrust::raw_pointer_cast((*d_first).base());
  auto d_first_ptr = const_cast<T*>(d_first_const_ptr);
  ::thrust::device_ptr<T> d_first_dev_ptr(d_first_ptr);

  if constexpr (::std::is_same_v<BinaryOp, ::thrust::plus<T>>) {
    if (d_init)
      *d_init += d_output;

    struct add_d_output {
      T d_output;
      add_d_output(T d) : d_output(d) {}
      CUDA_HOST_DEVICE void operator()(T& x) const { x += d_output; }
    };

    if (n > 0) {
      ::thrust::for_each(d_first_dev_ptr, d_first_dev_ptr + n,
                         add_d_output(d_output));
    }
```

Core Algorithm Implementations

- Transformation Operations
  1. thrust::transform
     - Element-wise operations with automatic differentiation
     - Generic functor support for arbitrary user transformations
     - Efficient GPU parallelization
  2. thrust::transform_reduce
     - Fused transformation and reduction
     - Critical for ML: dot products, norms, loss functions
     - Minimizes memory traffic between operations

Core Algorithm Implementations

- Transformation Operations
  1. thrust::transform

```
template <typename InputIt, typename OutputIt, typename UnaryOp>
void transform_pullback(InputIt first, InputIt last, OutputIt result,
                        UnaryOp op, OutputIt d_return, InputIt* d_first,
                        InputIt* d_last, OutputIt* d_result, UnaryOp* d_op) {
  size_t n = ::thrust::distance(first, last);
```

```
if constexpr (::std::is_same_v<UnaryOp, ::thrust::negate<Value>>) {
  struct grad_functor {
    CUDA_HOST_DEVICE void
    operator()(::thrust::tuple<Value&, Value&> t) const {
      ::thrust::get<0>(t) -= ::thrust::get<1>(t);
      ::thrust::get<1>(t) = 0;
    }
  };
  auto iter = ::thrust::make_zip_iterator(
      ::thrust::make_tuple(d_src_dev_ptr, d_dst_dev_ptr));
  ::thrust::for_each(iter, iter + n, grad_functor());
```

Core Algorithm Implementations

- Transformation Operations
    1. thrust::transform_reduce

```
// 1. Perform the forward transform to get intermediate values.
::thrust::device_vector<TransformedType> transformed_values(n);
::thrust::transform(first, last, transformed_values.begin(), unary_op);

// 2. Compute gradients for the intermediate transformed values by calling
// reduce_pullback.
::thrust::device_vector<TransformedType> d_transformed_values(n);
auto d_transformed_begin = d_transformed_values.begin();
auto d_transformed_end_dummy = d_transformed_values.end();

reduce_pullback(transformed_values.begin(), transformed_values.end(), init,
                binary_op, d_output, &d_transformed_begin,
                &d_transformed_end_dummy, d_init, d_binary_op);

// 3. Propagate gradients from the transformed values back to the original
// input.
::thrust::device_vector<TransformedType> d_result_dummy(n);
auto d_transformed_values_it = d_transformed_values.begin();
transform_pullback(first, last, transformed_values.begin(), unary_op,
                   d_result_dummy.begin(), // d_return dummy
                   d_first, d_last, &d_transformed_values_it, d_unary_op);
}
```

Advanced Operations

- Scan Operations (Prefix Sums)
  1. Inclusive and Exclusive Scans:
     - Fundamental building block for parallel algorithms
     - Applications: cumulative distributions, parallel scheduling, dynamic programming
     - Efficient parallel backward pass for gradient accumulation

  2. Technical Challenge: Output at position *i* depends on all inputs up to *i*

Advanced Operations

- Scan Operations (Prefix Sums)

```
if constexpr (::std::is_same_v<BinaryOp, ::thrust::plus<Value>>) {
  ::thrust::device_vector<Value> suffix_sums(n);

  ::thrust::inclusive_scan(::thrust::make_reverse_iterator(d_dst_dev_ptr + n),
                           ::thrust::make_reverse_iterator(d_dst_dev_ptr),
                           suffix_sums.begin(), op);

  ::thrust::transform(d_src_dev_ptr, d_src_dev_ptr + n,
                      ::thrust::make_reverse_iterator(suffix_sums.end()),
                      d_src_dev_ptr, ::thrust::plus<Value>());

  ::thrust::fill(d_dst_dev_ptr, d_dst_dev_ptr + n, Value(0));
```

Advanced Operations

- Sorting Primitives
  1. thrust::sort_by_key:
     - Sort key-value pairs while maintaining gradient flow
     - Forward pass records index permutation
     - Backward pass applies inverse permutation to gradients

Advanced Operations

- Sorting Primitives
  1. thrust::sort_by_key:

```
// Retrieve permutation mapping sorted position j -> original index i
auto& stack = detail::permutation_stack();
::thrust::device_vector<::std::size_t> perm = ::std::move(stack.back());
stack.pop_back();

// Build device pointers to adjoint buffers
auto d_vals_const_ptr = ::thrust::raw_pointer_cast((*d_values_first).base());
auto d_vals_ptr = const_cast<Value*>(d_vals_const_ptr);
::thrust::device_ptr<Value> d_vals(d_vals_ptr);

// Make a copy of current (sorted order) adjoints, then scatter-add into
// original positions and clear the sorted adjoints.
::thrust::device_vector<Value> dvals_tmp(n);
::thrust::copy(d_vals, d_vals + n, dvals_tmp.begin());
::thrust::fill(d_vals, d_vals + n, Value(0));

auto out_perm_begin =
    ::thrust::make_permutation_iterator(d_vals, perm.begin());
auto out_perm_end =
    ::thrust::make_permutation_iterator(d_vals, perm.begin() + n);
::thrust::transform(out_perm_begin, out_perm_end, dvals_tmp.begin(),
                    out_perm_begin, ::thrust::plus<Value>());
```

Advanced Operations

- Segmented Operations
  1. thrust::reduce_by_key:
     - Group-wise reductions (SQL-like GROUP BY on GPU)
     - Critical for batch processing in neural networks

  2. Segmented Scans:
     - Prefix sums within each segment
     - Complex gradient routing through irregular partition boundaries

Advanced Operations

- Segmented Operations
    1. thrust::exclusive_scan_by_key

```
if constexpr (::std::is_same_v<BinaryOp, ::thrust::plus<Value>> &&
              ::std::is_same_v<KeyEqual, ::thrust::equal_to<Key>>) {
  if (d_init) {
    *d_init +=
        ::thrust::reduce(d_dst_dev_ptr, d_dst_dev_ptr + n, Value(0), op);
  }

  ::thrust::device_vector<Value> suffix_sums(n);

  auto rev_keys_begin = ::thrust::make_reverse_iterator(keys_last);
  auto rev_keys_end = ::thrust::make_reverse_iterator(keys_first);
  auto rev_dy_begin = ::thrust::make_reverse_iterator(d_dst_dev_ptr + n);
  auto rev_dy_end = ::thrust::make_reverse_iterator(d_dst_dev_ptr);

  ::thrust::exclusive_scan_by_key(rev_keys_begin, rev_keys_end, rev_dy_begin,
                                  suffix_sums.begin(), Value(0),
                                  ::thrust::equal_to<Key>(), op);

  ::thrust::transform(d_src_dev_ptr, d_src_dev_ptr + n,
                      ::thrust::make_reverse_iterator(suffix_sums.end()),
                      d_src_dev_ptr, ::thrust::plus<Value>());

  ::thrust::fill(d_dst_dev_ptr, d_dst_dev_ptr + n, Value(0));
```

Demo: Logistic Regression

Idea: Classification using GPU acceleration.

Features:

- Logistic regression with gradient descent
- Cross-entropy loss function
- GPU Acceleration:
  - All operations on device memory
  - Thrust operations for vectorized math
- Automatic differentiation via Clad

Demo: Logistic Regression

Thrust Functions in Action

```
using Vec = thrust::device_vector<double>;
static inline double dot(const Vec& a, const Vec& b) {
    return thrust::inner_product(a.begin(), a.end(), b.begin(), 0.0);
}
static inline double sigmoid(double z) { return 1.0 / (1.0 + std::exp(-z)); }

// Minimal single-document logistic loss.
// x: (V), w: (V), y in {0,1}
double logistic_loss_single(const Vec& x, const Vec& w, double b, double y) {
    double logit = dot(x, w) + b;
    double p = sigmoid(logit);
    const double eps = 1e-9;
    return -y * std::log(p + eps) - (1.0 - y) * std::log(1.0 - p + eps);
}
```

Demo: Logistic Regression

Thrust Functions in Action

```
static inline void zero(Vec& v) { thrust::fill(v.begin(), v.end(), 0.0); }

static inline void sgd_step(Vec& w, const Vec& dw, double& b, double db,
                            double lr) {
  thrust::transform(
      w.begin(), w.end(), dw.begin(), w.begin(),
      [=] __device__(double wi, double gi) { return wi - lr * gi; });
  b -= lr * db;
}
```

```
auto grad = clad::gradient(logistic_loss_single);
Vec dx(V), dw(V);
double db = 0.0;
double dy = 0.0;

grad.execute(x, w, b, y, &dx, &dw, &db, &dy);

// Run SGD on a single document.
{
  std::cout << "\nRunning SGD on a single document..." << std::endl;
  double lr = 0.1;
  for (int t = 0; t < 10; ++t) {
      zero(dw);
      zero(db);
      dy = 0.0;
      grad.execute(x, w, b, y, &dx, &dw, &db, &dy);
      sgd_step(w, dw, b, db, lr);
      double loss = logistic_loss_single(x, w, b, y);
      std::cout << "iter " << t << ": loss=" << loss << std::endl;
  }
  double loss = logistic_loss_single(x, w, b, y);
  std::cout << "Loss: " << loss << std::endl;
}
```

Challenges & Solutions

1. **GPU Memory Errors**
    - **Problem:** Tracing memory access violations within the CUDA/Thrust environment was complex.
    - **Solution:** Used `compute-sanitizer` and careful GPU pointer management to resolve memory errors.

2. **Mathematical Edge Cases**
    - **Problem:** Derivatives undefined for certain operations (e.g., multiply by zero)
    - **Solution:** Implemented logic to count zeros and correctly handle the gradient for single and multiple zero-value inputs.

3. **Correctness Validation**
    - **Problem:** Verifying GPU-accelerated derivatives
    - **Solution:** Finite difference comparison, Comprehensive unit tests and Integration tests with real demos

Future Goals

- **Supporting more Thrust primitives**:
    - Finalize support for the remaining Thrust algorithms.
    - Expand the support for the functor handling.

- **Testing Use case**:
    - Develop more advanced, real-world examples, such as in neural network training.

Thanks!