Implement and improve an efficient, layered tape with prefetching capabilities

•••

Aditi Milind Joshi

GSoC 2025: Wrap-Up Presentation

The Tape

In clad, the tape is a stack-like data structure that stores intermediate values in reverse mode AD during the forward pass for use during the backward (gradient) pass.

Old Implementation

Clad used a monolithic memory buffer, which, when full while pushing, reallocated double the capacity.

```
/// Move values from old to new storage
CUDA_HOST_DEVICE T* AllocateRawStorage(std::size_t _capacity) {
  #ifdef __CUDACC__
    // Allocate raw storage (without calling constructors of T) of new capacity.
    T* new_data = static_cast<T*>(::operator new(_capacity * sizeof(T)));
  #else
    T *new_data =
      static_cast<T *>(::operator new(_capacity * sizeof(T), std::nothrow));
  #endif
  return new_data;
}

/// Add new value of type T constructed from args to the end of the tape.
template <typename... ArgsT>
CUDA_HOST_DEVICE void emplace_back(ArgsT&&... args) {
  if (_size >= _capacity)
    grow();
  ::new (const_cast<void*>(static_cast<const volatile void*>(end())))
      T(std::forward<ArgsT>(args)...);
  _size += 1;
}
```

```
private:
  // Copies the data from a storage to another.
  // Implementation taken from std::uninitialized_copy
  template <class InputIt, class NoThrowForwardIt>
  CUDA_HOST_DEVICE void MoveData(InputIt first, InputIt last,
                                 NoThrowForwardIt d_first) {
    NoThrowForwardIt current = d_first;
    // We specifically add and remove the CV qualifications here so that
    // cases where NoThrowForwardIt is CV qualified, we can still do the
    // allocation properly.
    for (; first != last; ++first, (void)++current) {
      ::new (const_cast<void*>(
          static_cast<const volatile void*>(clad_addressof(*current))))
        T(std::move(*first));
    }
  }
  /// Initial capacity (allocated whenever a value is pushed into empty tape).
  constexpr static std::size_t _init_capacity = 32;
  CUDA_HOST_DEVICE void grow() {
    // If empty, use initial capacity.
    if (!_capacity)
      _capacity = _init_capacity;
    else
      // Double the capacity on each reallocation.
      _capacity *= 2;
    T* new_data = AllocateRawStorage(_capacity);

    if (!new_data) {
      // clean up the memory mess just in case!
      destroy(begin(), end());
      printf("Allocation failure during tape resize! Aborting.\n");
      trap(EXIT_FAILURE);
    }
```

Midterm Progress - Slab-based Tape with SBO

- The new tape follows a slab-based structure with small buffer optimization.
- Slab size and buffer size are configurable template parameters.
- Slab size 1024 and buffer size 64 performed the best and were set as default values.

```
/// A dynamic slab-based vector-like container with Small Buffer Optimization
/// (SBO), primarily used for storing values in reverse-mode AD. Stores elements
/// in a static buffer first, then falls back to dynamically allocated linked
/// slabs if capacity exceeds SBO.
template <typename T, std::size_t SBO_SIZE = 64, std::size_t SLAB_SIZE = 1024>
class tape_impl {
  /// A block of contiguous storage allocated dynamically when SBO capacity is
  /// exceeded.
  struct Slab {
    // std::aligned_storage_t<sizeof(T), alignof(T)> raw_data[SLAB_SIZE];
    // For now use the implementation below as above implementation is not
    // supported by c++11
    alignas(T) char raw_data[SLAB_SIZE * sizeof(T)]{};
    Slab* next;
    CUDA_HOST_DEVICE Slab() : next(nullptr) {}
    CUDA_HOST_DEVICE T* elements() {
#if __cplusplus >= 201703L
      return std::launder(reinterpret_cast<T*>(raw_data));
#else
      return reinterpret_cast<T*>(raw_data);
#endif
    }
  };

  // std::aligned_storage_t<sizeof(T), alignof(T)> m_static_buffer[SBO_SIZE];
  // For now use the implementation below as above implementation is not
  // supported by c++11
  alignas(T) char m_static_buffer[SBO_SIZE * sizeof(T)]{};
  bool m_using_sbo = true;

  Slab* m_head = nullptr;
  std::size_t m_size = 0;

  CUDA_HOST_DEVICE T* sbo_elements() {
#if __cplusplus >= 201703L
    return std::launder(reinterpret_cast<T*>(m_static_buffer));
#else
    return reinterpret_cast<T*>(m_static_buffer);
#endif
  }

  CUDA_HOST_DEVICE const T* sbo_elements() const {
#if __cplusplus >= 201703L
    return std::launder(reinterpret_cast<const T*>(m_static_buffer));
#else
    return reinterpret_cast<const T*>(m_static_buffer);
#endif
  }
```

```
/// Add new value of type T constructed from args to the end of the tape.
template <typename... ArgsT>
CUDA_HOST_DEVICE void emplace_back(ArgsT&&... args) {
  if (m_size < SBO_SIZE) {
    // Store in SBO buffer
    ::new (const_cast<void*>(static_cast<const volatile void*>(
        sbo_elements() + m_size))) T(std::forward<ArgsT>(args)...);
  } else {
    // Transition to dynamic storage if needed
    if (m_using_sbo)
      m_using_sbo = false;

    // Allocate new slab if required
    if ((m_size - SBO_SIZE) % SLAB_SIZE == 0) {
      Slab* new_slab = new Slab();
      if (!m_head) {
        m_head = new_slab;
      } else {
        Slab* last = m_head;
        while (last->next)
          last = last->next;
        last->next = new_slab;
      }
    }

    // Find correct slab for element
    Slab* slab = m_head;
    std::size_t idx = (m_size - SBO_SIZE) / SLAB_SIZE;
    while (idx--)
      slab = slab->next;

    // Construct element in-place
    ::new (const_cast<void*>(static_cast<const volatile void*>(
        slab->elements() + ((m_size - SBO_SIZE) % SLAB_SIZE))))
        T(std::forward<ArgsT>(args)...);
  }
  m_size++;
}
```

Fixes in Benchmarks & Tape Improvements

Benchmark Fixes

- Removed Iterations(1) to get better estimate of the benchmarks.
- Fixed Memory Manager Counters.
- Added DoNotOptimize() to prevent compiler from optimizing out the pop function.

Tape Improvements

- Added Tail Pointer reducing the push and pop function runs from O(n) to O(1).
- Changed tape to doubly linked list.
- Reused slabs which were previously allocated.

Updated Code

```
/// A dynamic slab-based vector-like container with Small Buffer Optimization
/// (SBO), primarily used for storing values in reverse-mode AD. Stores elements
/// in a static buffer first, then falls back to dynamically allocated linked
/// slabs if capacity exceeds SBO.
template <typename T, std::size_t SBO_SIZE = 64, std::size_t SLAB_SIZE = 1024,
          bool is_multithread = false>
class tape_impl {
  /// A block of contiguous storage allocated dynamically when SBO capacity is
  /// exceeded.
  struct Slab {
    // std::aligned_storage_t<sizeof(T), alignof(T)> raw_data[SLAB_SIZE];
    // For now use the implementation below as above implementation is not
    // supported by c++11
    alignas(T) char raw_data[SLAB_SIZE * sizeof(T)];
    Slab* prev;
    Slab* next;
    CUDA_HOST_DEVICE Slab() : prev(nullptr), next(nullptr) {}
    CUDA_HOST_DEVICE T* elements() {
#if __cplusplus >= 201703L
      return std::launder(reinterpret_cast<T*>(raw_data));
#else
      return reinterpret_cast<T*>(raw_data);
#endif
    }
  };

  // std::aligned_storage_t<sizeof(T), alignof(T)> m_static_buffer[SBO_SIZE];
  // For now use the implementation below as above implementation is not
  // supported by c++11
  alignas(T) char m_static_buffer[SBO_SIZE * sizeof(T)];

  Slab* m_head = nullptr;
  Slab* m_tail = nullptr;
  std::size_t m_size = 0;
  std::size_t m_capacity = SBO_SIZE;
```

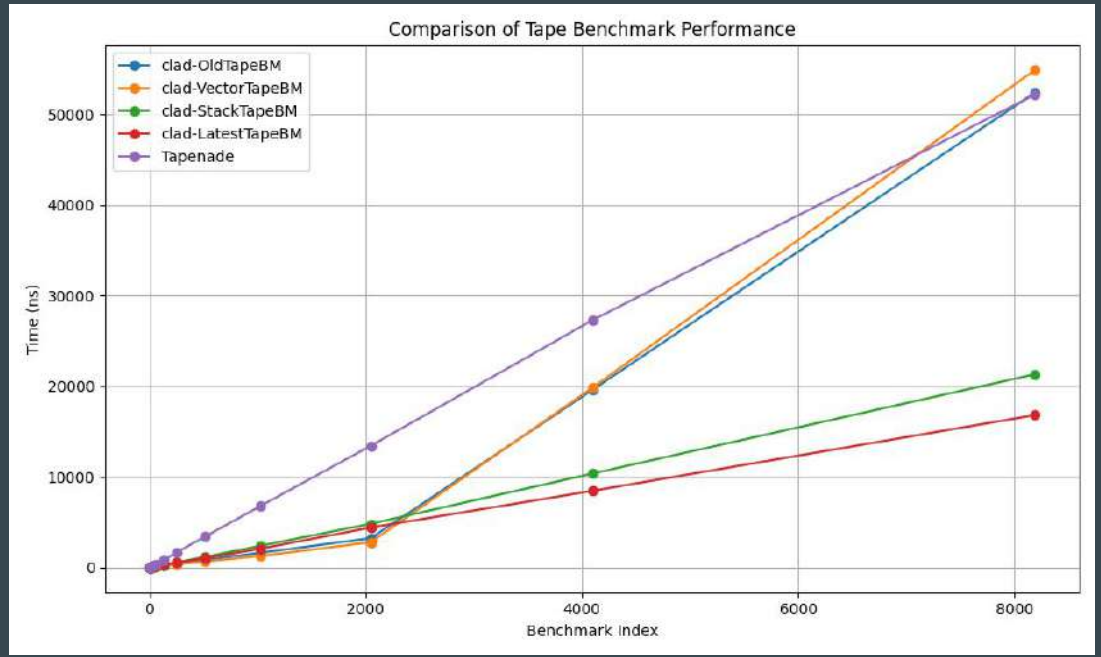
```
/// Add new value of type T constructed from args to the end of the tape.
template <typename... ArgsT>
CUDA_HOST_DEVICE void emplace_back(ArgsT&&... args) {
  if (m_size < SBO_SIZE) {
    // Store in SBO buffer
    ::new (const_cast<void*>(static_cast<const volatile void*>(
        sbo_elements() + m_size))) T(std::forward<ArgsT>(args)...);
  } else {
    const auto offset = (m_size - SBO_SIZE) % SLAB_SIZE;
    // Allocate new slab if required
    if (!offset) {
      if (m_size == m_capacity) {
        Slab* new_slab = new Slab();
        if (!m_head)
          m_head = new_slab;
        else {
          m_tail->next = new_slab;
          new_slab->prev = m_tail;
        }
        m_capacity += SLAB_SIZE;
      }
      if (m_size == SBO_SIZE)
        m_tail = m_head;
      else
        m_tail = m_tail->next;
    }

    // Construct element in-place
    ::new (const_cast<void*>(static_cast<const volatile void*>(
        m_tail->elements() + offset))) T(std::forward<ArgsT>(args)...);
  }
  m_size++;
}
```

```
/// Remove the last value from the tape.
CUDA_HOST_DEVICE void pop_back() {
  assert(m_size);
  m_size--;
  if (m_size < SBO_SIZE)
    destroy_element(sbo_elements() + m_size);
  else {
    std::size_t offset = (m_size - SBO_SIZE) % SLAB_SIZE;
    destroy_element(m_tail->elements() + offset);
    if (offset == 0) {
      if (m_tail != m_head)
        m_tail = m_tail->prev;
    }
  }
}
```

New Benchmarks

- Tested against the old tape, std::vector, std::stack and tapenade.



Thread Safety

- Added thread-safe tape access functions with mutex locking mechanism to allow concurrent access.

```
/// Thread safe tape access functions with mutex locking mechanism
#ifndef __CUDACC__
/// Add value to the end of the tape, return the same value.
template <typename T, std::size_t SBO_SIZE = 64, std::size_t SLAB_SIZE = 1024,
          typename... ArgsT>
T push(tape<T, SBO_SIZE, SLAB_SIZE, /*is_multithreaded=*/true>& to,
       ArgsT&&... val) {
  std::lock_guard<std::mutex> lock(to.mutex());
  to.emplace_back(std::forward<ArgsT>(val)...);
  return to.back();
}

/// A specialization for C arrays
template <typename T, typename U, size_t N, std::size_t SBO_SIZE = 64,
          std::size_t SLAB_SIZE = 1024>
void push(tape<T[N], SBO_SIZE, SLAB_SIZE, /*is_multithreaded=*/true>& to,
          const U& val) {
  std::lock_guard<std::mutex> lock(to.mutex());
  to.emplace_back();
  std::copy(std::begin(val), std::end(val), std::begin(to.back()));
}

/// Remove the last value from the tape, return it.
template <typename T, std::size_t SBO_SIZE = 64, std::size_t SLAB_SIZE = 1024>
T pop(tape<T, SBO_SIZE, SLAB_SIZE, /*is_multithreaded=*/true>& to) {
  std::lock_guard<std::mutex> lock(to.mutex());
  T val = std::move(to.back());
  to.pop_back();
  return val;
}

/// A specialization for C arrays
template <typename T, std::size_t N, std::size_t SBO_SIZE = 64,
          std::size_t SLAB_SIZE = 1024>
void pop(tape<T[N], SBO_SIZE, SLAB_SIZE, /*is_multithreaded=*/true>& to) {
  std::lock_guard<std::mutex> lock(to.mutex());
  to.pop_back();
}

/// Access return the last value in the tape.
template <typename T, std::size_t SBO_SIZE = 64, std::size_t SLAB_SIZE = 1024>
T& back(tape<T, SBO_SIZE, SLAB_SIZE, /*is_multithreaded=*/true>& of) {
  std::lock_guard<std::mutex> lock(of.mutex());
  return of.back();
}
#endif
```

Ongoing Progress - Multilayer Tape

Multilayer Tape Storage

- Offloading mechanism to offload slabs to disk and load slabs from disk to memory.
- Keep only the last N slabs in memory at a time and offload the rest.
- One slab space where slabs are loaded if element to be loaded is not in memory.

Progress

Current Progress

- Benchmark script to compare two revisions (PR #1394)
- Modified tape structure to slab-based (PR #1404)
- Added small buffer optimization (PR #1404)
- Enhanced benchmarks (PR #1404 and PR #1533)
- Made improvements to tape(PR #1531 and PR #1633)
- Added thread safety (PR #1470)

Thank You!