GSoC 2025
Progress Update

Implement and improve an efficient, layered tape with prefetching capabilities

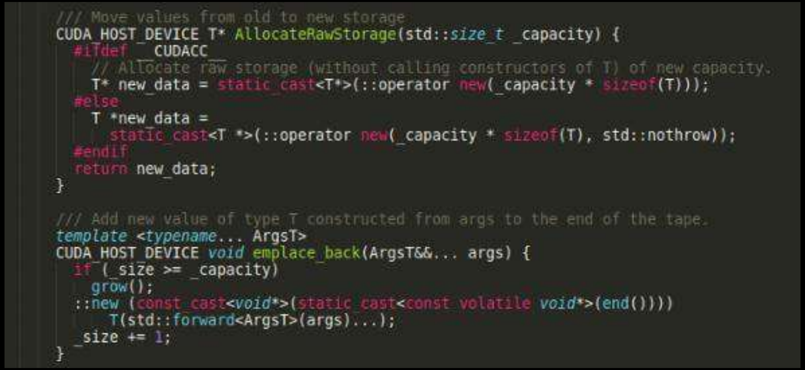
Aditi Milind Joshi

Tape in Clad

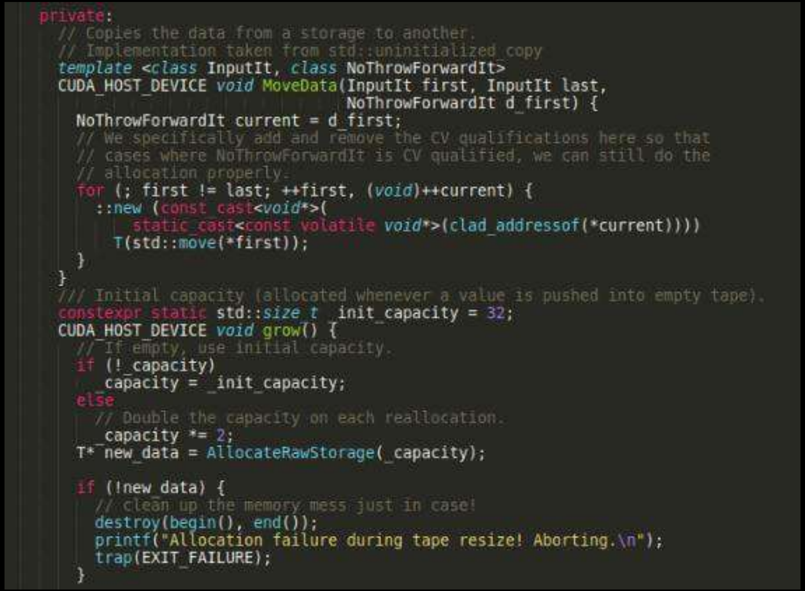
The tape is a stack-like data structure that stores intermediate values in reverse mode AD during the forward pass for use during the backward (gradient) pass

Previous Implementation of Tape

Clad used a monolithic memory buffer. If the tape was full while pushing then it reallocated double the capacity.



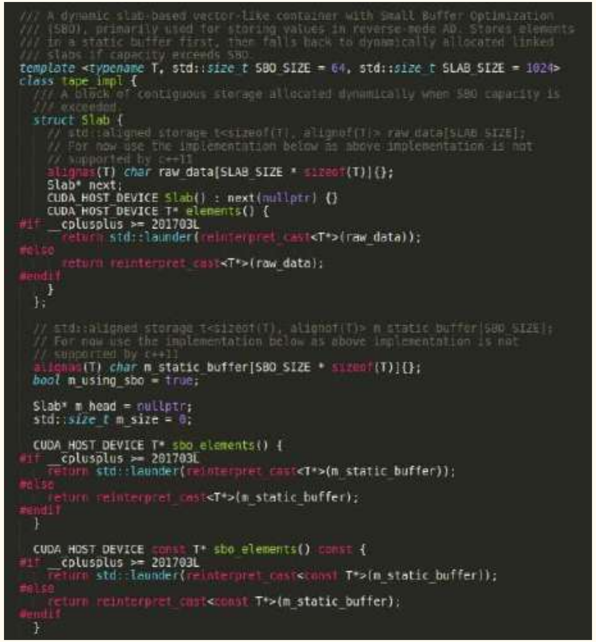
```cpp
/// Move values from old to new storage
CUDA_HOST_DEVICE T* AllocateRawStorage(std::size_t _capacity) {
  #ifdef __CUDACC__
    // Allocate raw storage (without calling constructors of T) of new capacity.
    T* new_data = static_cast<T*>(::operator new(_capacity * sizeof(T)));
  #else
    T *new_data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



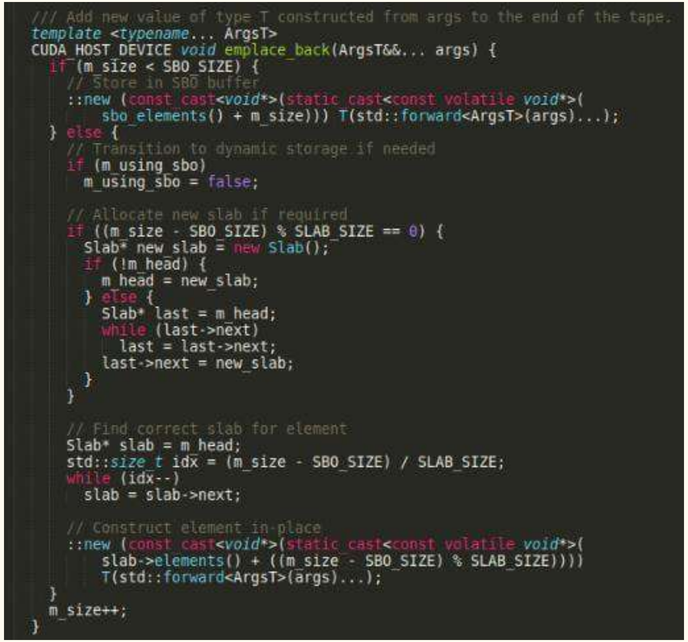
```cpp
private:
  // Copies the data from a storage to another.
  // Implementation taken from std::uninitialized_copy
  template <class InputIt, class NoThrowForwardIt>
  CUDA_HOST_DEVICE void MoveData(InputIt first, InputIt last,
                                 NoThrowForwardIt d_first) {
    NoThrowForwardIt 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);
    }
```

New Implementation of Tape

The new tape follows a slab-based structure with small buffer optimization. The slab size and buffer size are configurable template parameters with default values.



```cpp
/// 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
  }
```

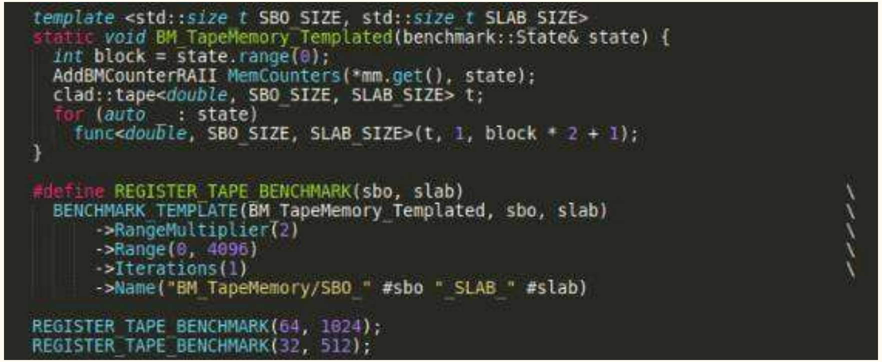


```cpp
/// Add new value of type T constructed from args to the end of the tape.
template <typename... ArgsT>
CUDA_HOST_DEVICE void emplace_back(ArgsT&&... args) {
  if (m_size < SBO_SIZE) {
    // Store in SBO buffer
    ::new (const_cast<void*>(static_cast<const volatile void*>(
        sbo_elements()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

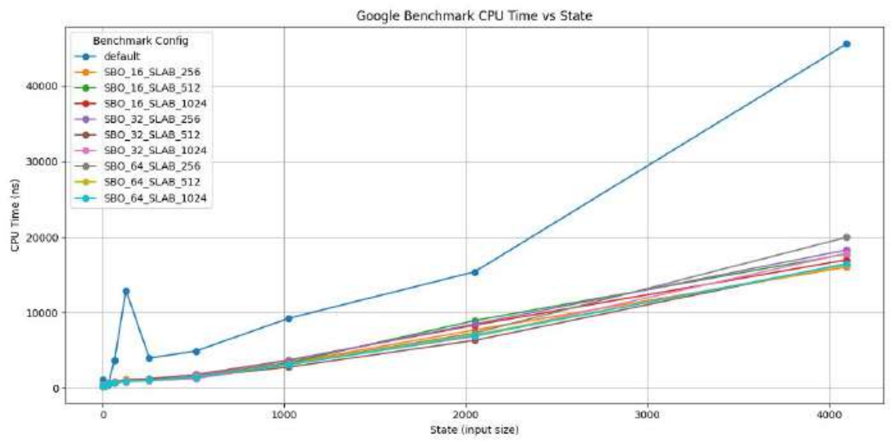
Enhanced Benchmarks

Configurable tape memory benchmarks added to test different configurations and find optimal slab and buffer size

Slab size 64 and buffer size 1024 performed the best consistently over many runs



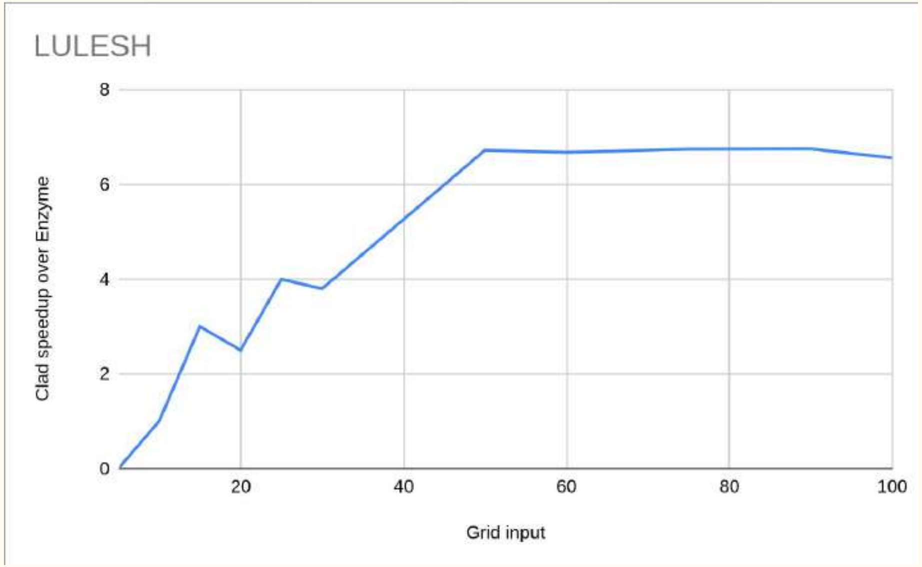
```
template <std::size_t SBO_SIZE, std::size_t SLAB_SIZE>
static void BM_TapeMemory_Templated(benchmark::State& state) {
  int block = state.range(0);
  AddBMCounterRAII MemCounters(*mm.get(), state);
  clad::tape<double, SBO_SIZE, SLAB_SIZE> t;
  for (auto _ : state)
    func<double, SBO_SIZE, SLAB_SIZE>(t, 1, block * 2 + 1);
}

#define REGISTER_TAPE_BENCHMARK(sbo, slab)                                     \
  BENCHMARK_TEMPLATE(BM_TapeMemory_Templated, sbo, slab)                       \
      ->RangeMultiplier(2)                                                     \
      ->Range(0, 4096)                                                         \
      ->Iterations(1)                                                          \
      ->Name("BM_TapeMemory/SBO_" #sbo "_SLAB_" #slab)

REGISTER_TAPE_BENCHMARK(64, 1024);
REGISTER_TAPE_BENCHMARK(32, 512);
```



Lulesh Benchmarks

grid size = 50: clad is 6x faster than enzyme

grid size = 100: clad is 8.6x faster than enzyme



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)

Future Work

- Add thread safety to tape
- Add offloading mechanism
- Implement CPU-GPU transfer (Stretch goal)
- Implement Checkpointing (Stretch goal)

Thank You!