Program Listing for File codecs.hpp

Return to documentation for file (include/holoscan/core/codecs.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2023-2024 NVIDIA CORPORATION & AFFILIATES. All rights reserved. * SPDX-License-Identifier: Apache-2.0 * * Licensed under the Apache License, Version 2.0 (the "License"); * you may not use this file except in compliance with the License. * You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License. */ #ifndef HOLOSCAN_CORE_CODECS_HPP #define HOLOSCAN_CORE_CODECS_HPP #include <complex> #include <cstdint> #include <functional> #include <memory> #include <string> #include <type_traits> #include <typeindex> #include <typeinfo> #include <unordered_map> #include <utility> #include <vector> #include "./common.hpp" #include "./endpoint.hpp" #include "./errors.hpp" #include "./expected.hpp" #include "./type_traits.hpp" #include "gxf/core/expected.hpp" #include "gxf/serialization/serialization_buffer.hpp" *// Note: Currently the GXF UCX extension transmits using little-endian byte order. All hardware // supported by Holoscan SDK is also little endian. To support big endian platforms, we // would need to update the codecs in this file to properly handle endianness.* namespace holoscan { *// codec template struct so users can define their own codec class // i.e. following the design of yaml-cpp's YAML::convert* template <typename T> struct codec; *// Codec type 1: trivial binary types (float, etc.) // // Types that can be serialized by writing binary data where the number // of bytes to be written is based on sizeof(typeT).* template <typename typeT> static inline expected<size_t, RuntimeError> serialize_trivial_type(const typeT& value, Endpoint* endpoint) { return endpoint->write_trivial_type(&value); } template <typename typeT> static inline expected<typeT, RuntimeError> deserialize_trivial_type(Endpoint* endpoint) { typeT encoded; auto maybe_value = endpoint->read_trivial_type(&encoded); if (!maybe_value) { return forward_error(maybe_value); } return encoded; } *// TODO: currently not handling integer types separately* template <typename typeT> struct codec { static expected<size_t, RuntimeError> serialize(const typeT& value, Endpoint* endpoint) { return serialize_trivial_type<typeT>(value, endpoint); } static expected<typeT, RuntimeError> deserialize(Endpoint* endpoint) { return

deserialize_trivial_type<typeT>(endpoint); } }; *// Codec type 3: basic container types // // For vectorT types storing multiple items contiguously in memory // (e.g. std::vector, std::array, std::string). // // It requires that vectorT have methods size(), data(), and operator[]* #pragma pack(push, 1) struct ContiguousDataHeader { size_t size; uint8_t bytes_per_element; }; #pragma pack(pop) template <typename vectorT> static inline expected<size_t, RuntimeError> serialize_binary_blob(const vectorT& data, Endpoint* endpoint) { ContiguousDataHeader header; header.size = data.size(); header.bytes_per_element = header.size > 0 ? sizeof(data[0]) : 1; auto size = endpoint->write_trivial_type<ContiguousDataHeader>(&header); if (!size) { return forward_error(size); } auto size2 = endpoint->write(data.data(), header.size * header.bytes_per_element); if (!size2) { return forward_error(size2); } return size.value() + size2.value(); } template <typename vectorT> static inline expected<vectorT, RuntimeError> deserialize_binary_blob(Endpoint* endpoint) { ContiguousDataHeader header; auto header_size = endpoint->read_trivial_type<ContiguousDataHeader>(&header); if (!header_size) { return forward_error(header_size); } vectorT data; data.resize(header.size); auto result = endpoint->read(data.data(), header.size * header.bytes_per_element); if (!result) { return forward_error(result); } return data; } *// codec for vector of trivially serializable typeT* template <typename typeT> struct codec<std::vector<typeT>> { static expected<size_t, RuntimeError> serialize(const std::vector<typeT>& value, Endpoint* endpoint) { return serialize_binary_blob<std::vector<typeT>>(value, endpoint); } static expected<std::vector<typeT>, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_binary_blob<std::vector<typeT>>(endpoint); } }; *// deserialize_array is exactly like deserialize_binary_blob, but without a call to resize()* template <typename arrayT> static inline expected<arrayT, RuntimeError> deserialize_array(Endpoint* endpoint) { ContiguousDataHeader header; auto header_size = endpoint->read_trivial_type<ContiguousDataHeader>(&header); if (!header_size) { return forward_error(header_size); } arrayT data; auto result = endpoint->read(data.data(), header.size * header.bytes_per_element); if (!result) { return forward_error(result); } return data; } *// codec for array of trivially serializable typeT and size N* template <typename typeT, size_t N> struct codec<std::array<typeT, N>> { static expected<size_t, RuntimeError> serialize(const std::array<typeT, N>& value, Endpoint* endpoint) { return serialize_binary_blob<std::array<typeT, N>>(value, endpoint); } static expected<std::array<typeT, N>, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_array<std::array<typeT, N>>

(endpoint); } }; *// codec for std::string* template <> struct codec<std::string> { static expected<size_t, RuntimeError> serialize(const std::string& value, Endpoint* endpoint) { return serialize_binary_blob<std::string>(value, endpoint); } static expected<std::string, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_binary_blob<std::string>(endpoint); } }; *// will hold Python cloudpickle strings in this container to differentiate from std::string* struct CloudPickleSerializedObject { std::string serialized; }; *// codec for CloudPickleSerializedObject* template <> struct codec<CloudPickleSerializedObject> { static expected<size_t, RuntimeError> serialize(const CloudPickleSerializedObject& value, Endpoint* endpoint) { return serialize_binary_blob<std::string> (value.serialized, endpoint); } static expected<CloudPickleSerializedObject, RuntimeError> deserialize(Endpoint* endpoint) { auto maybe_string = deserialize_binary_blob<std::string>(endpoint); if (!maybe_string) { return forward_error(maybe_string); } CloudPickleSerializedObject cloudpickle_obj{std::move(maybe_string.value())}; return cloudpickle_obj; } }; *// Codec type 4: serialization of std::vector<bool> only // // Performs bit-packing/unpacking to/from uint8_t type for more efficient serialization. // codec of std::vector<bool>* template <> struct codec<std::vector<bool>> { static expected<size_t, RuntimeError> serialize(const std::vector<bool>& data, Endpoint* endpoint) { size_t total_bytes = 0; size_t num_bits = data.size(); size_t num_bytes = (num_bits + 7) / 8; *// the number of bytes needed to store the bits* auto size = endpoint->write_trivial_type<size_t>(&num_bits); if (!size) { return forward_error(size); } total_bytes += size.value(); std::vector<uint8_t> packed_data(num_bytes, 0); *// Create a vector to store the packed data* for (size_t i = 0; i < num_bits; ++i) { if (data[i]) { packed_data[i / 8] |= (1 << (i % 8)); *// Pack the bits into the bytes* } } auto result = endpoint->write(packed_data.data(), packed_data.size()); if (!result) { return forward_error(result); } total_bytes += result.value(); return total_bytes; } static expected<std::vector<bool>, RuntimeError> deserialize(Endpoint* endpoint) { size_t num_bits; auto size = endpoint->read_trivial_type<size_t>(&num_bits); if (!size) { return forward_error(size); } size_t num_bytes = (num_bits + 7) / 8; *// Calculate the number of bytes needed to store the bits* std::vector<uint8_t> packed_data(num_bytes, 0); *// Create a vector to store the packed data* auto result = endpoint->read(packed_data.data(), packed_data.size()); if (!result) { return forward_error(result); } std::vector<bool> data(num_bits, false); *// Create a vector to store the unpacked data* for (size_t i = 0; i < num_bits; ++i) { if (packed_data[i / 8] & (1 << (i % 8))) { *// Unpack the bits from the bytes* data[i] = true; } } return data; } }; *//*

Codec type 5: serialization of nested container types // e.g. std::vector<std::vector<float>>, std::vector<std::string> // template <typename typeT> static inline expected<size_t, RuntimeError> serialize_vector_of_vectors(const typeT& vectors, Endpoint* endpoint) { size_t total_size = 0; *// header is just the total number of vectors* size_t num_vectors = vectors.size(); auto size = endpoint->write_trivial_type<size_t>(&num_vectors); if (!size) { return forward_error(size); } total_size += size.value(); using vectorT = typename typeT::value_type; *// now transmit each individual vector* for (const auto& vector : vectors) { size = codec<vectorT>::serialize(vector, endpoint); if (!size) { return forward_error(size); } total_size += size.value(); } return total_size; } template <typename typeT> static inline expected<typeT, RuntimeError> deserialize_vector_of_vectors(Endpoint* endpoint) { size_t num_vectors; auto size = endpoint->read_trivial_type<size_t>(&num_vectors); if (!size) { return forward_error(size); } using vectorT = typename typeT::value_type; std::vector<vectorT> data; data.reserve(num_vectors); for (size_t i = 0; i < num_vectors; i++) { auto vec = codec<vectorT>::deserialize(endpoint); if (!vec) { return forward_error(vec); } data.push_back(vec.value()); } return data; } template <typename typeT> struct codec<std::vector<std::vector<typeT>>> { static expected<size_t, RuntimeError> serialize(const std::vector<std::vector<typeT>>& value, Endpoint* endpoint) { return serialize_vector_of_vectors<std::vector<std::vector<typeT>>>(value, endpoint); } static expected<std::vector<std::vector<typeT>>, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_vector_of_vectors<std::vector<std::vector<typeT>>>(endpoint); } }; template <> struct codec<std::vector<std::string>> { static expected<size_t, RuntimeError> serialize(const std::vector<std::string>& value, Endpoint* endpoint) { return serialize_vector_of_vectors<std::vector<std::string>>(value, endpoint); } static expected<std::vector<std::string>, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_vector_of_vectors<std::vector<std::string>>(endpoint); } }; *// codec for shared_ptr types // Serializes the contents of the shared_ptr. On deserialize, a new shared_ptr to the deserialized // value is returned.* template <typename typeT> struct codec<std::shared_ptr<typeT>> { static expected<size_t, RuntimeError> serialize(std::shared_ptr<typeT> value, Endpoint* endpoint) { return codec<typeT>::serialize(*value, endpoint); } static expected<std::shared_ptr<typeT>, RuntimeError> deserialize(Endpoint* endpoint) { auto value = codec<typeT>::deserialize(endpoint); if (!value) { return forward_error(value); }

```
return std::make_shared<typeT>(value.value()); } }; } // namespace holoscan #endif/*
HOLOSCAN_CORE_CODECS_HPP */
```