Program Listing for File cuda_stream_handler.hpp

Return to documentation for file (gxf_extensions/utils/cuda_stream_handler.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2022 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 GXF_EXTENSIONS_UTILS_CUDA_STREAM_HANDLER_HPP #define GXF_EXTENSIONS_UTILS_CUDA_STREAM_HANDLER_HPP #include <utility> #include <vector> #include "gxf/cuda/cuda_stream.hpp" #include "gxf/cuda/cuda_stream_id.hpp" #include "gxf/cuda/cuda_stream_pool.hpp" namespace nvidia { namespace holoscan { class CudaStreamHandler { public: ~CudaStreamHandler() { for (auto&& event : cuda_events_) { const cudaError_t result = cudaEventDestroy(event); if (cudaSuccess != result) { GXF_LOG_ERROR("Failed to destroy CUDA event: %s", cudaGetErrorString(result)); } } cuda_events_.clear(); } gxf::Expected<void> registerInterface(gxf::Registrar* registrar, bool required = false) { return registrar->parameter(cuda_stream_pool_, "cuda_stream_pool", "CUDA Stream Pool", "Instance of gxf::CudaStreamPool.", gxf::Registrar::NoDefaultParameter(), required ? GXF_PARAMETER_FLAGS_NONE : GXF_PARAMETER_FLAGS_OPTIONAL); } gxf_result_t fromMessage(gxf_context_t context, const nvidia::gxf::Expected<nvidia::gxf::Entity>& message) { *// if the message contains a stream use this* const auto maybe_cuda_stream_id = message.value().get<gxf::CudaStreamId>(); if (maybe_cuda_stream_id) { const auto maybe_cuda_stream_handle = gxf::Handle<gxf::CudaStream>::Create(context, maybe_cuda_stream_id.value()->stream_cid); if (maybe_cuda_stream_handle) { message_cuda_stream_handle_ = maybe_cuda_stream_handle.value(); } } else { *// if no stream had been found, allocate a stream and use that* gxf_result_t result = allocateInternalStream(); if (result != GXF_SUCCESS) { return result; } message_cuda_stream_handle_ = cuda_stream_handle_; } return GXF_SUCCESS; } gxf_result_t fromMessages(gxf_context_t context, const std::vector<nvidia::gxf::Entity>& messages) { const gxf_result_t result =

allocateInternalStream(); if (result != GXF_SUCCESS) { return result; } if (!cuda_stream_handle_) { *// if no CUDA stream can be allocated because no stream pool is set, then don't sync // with incoming streams. CUDA operations of this operator will use the default stream // which sync with all other streams by default.* return GXF_SUCCESS; } *// iterate through all messages and use events to chain incoming streams with the internal // stream* auto event_it = cuda_events_.begin(); for (auto& msg : messages) { const auto maybe_cuda_stream_id = msg.get<gxf::CudaStreamId>(); if (maybe_cuda_stream_id) { const auto maybe_cuda_stream_handle = gxf::Handle<gxf::CudaStream>::Create(context, maybe_cuda_stream_id.value()->stream_cid); if (maybe_cuda_stream_handle) { const cudaStream_t cuda_stream = maybe_cuda_stream_handle.value()->stream().value(); cudaError_t result; *// allocate a new event if needed* if (event_it == cuda_events_.end()) { cudaEvent_t cuda_event; result = cudaEventCreateWithFlags(&cuda_event, cudaEventDisableTiming); if (cudaSuccess != result) { GXF_LOG_ERROR("Failed to create input CUDA event: %s", cudaGetErrorString(result)); return GXF_FAILURE; } cuda_events_.push_back(cuda_event); event_it = cuda_events_.end(); --event_it; } result = cudaEventRecord(*event_it, cuda_stream); if (cudaSuccess != result) { GXF_LOG_ERROR("Failed to record event for message stream: %s", cudaGetErrorString(result)); return GXF_FAILURE; } result = cudaStreamWaitEvent(cuda_stream_handle_->stream().value(), *event_it); if (cudaSuccess != result) { GXF_LOG_ERROR("Failed to record wait on message event: %s", cudaGetErrorString(result)); return GXF_FAILURE; } ++event_it; } } } message_cuda_stream_handle_ = cuda_stream_handle_; return GXF_SUCCESS; } gxf_result_t toMessage(nvidia::gxf::Expected<nvidia::gxf::Entity>& message) { if (message_cuda_stream_handle_) { const auto maybe_stream_id = message.value().add<gxf::CudaStreamId>(); if (!maybe_stream_id) { GXF_LOG_ERROR("Failed to add CUDA stream id to output message."); return gxf::ToResultCode(maybe_stream_id); } maybe_stream_id.value()->stream_cid = message_cuda_stream_handle_.cid(); } return GXF_SUCCESS; } gxf::Handle<gxf::CudaStream> getStreamHandle() { *// If there is a message stream handle, return this* if (message_cuda_stream_handle_) { return message_cuda_stream_handle_; } *// else allocate an internal CUDA stream and return it* allocateInternalStream(); return cuda_stream_handle_; } cudaStream_t getCudaStream() { const gxf::Handle<gxf::CudaStream> cuda_stream_handle = getStreamHandle(); if (cuda_stream_handle) { return cuda_stream_handle->stream().value(); } if (!default_stream_warning_) { default_stream_warning_ = true;

```
GXF_LOG_WARNING( "Parameter `cuda_stream_pool` is not set, using the default CUDA stream for CUDA " "operations."); } return cudaStreamDefault; } private: gxf_result_t allocateInternalStream() { // Create the CUDA stream if it does not yet exist. if (!cuda_stream_handle_) { const auto cuda_stream_pool = cuda_stream_pool_.try_get(); if (cuda_stream_pool) { // allocate a stream auto maybe_stream = cuda_stream_pool.value()->allocateStream(); if (!maybe_stream) { GXF_LOG_ERROR("Failed to allocate CUDA stream"); return gxf::ToResultCode(maybe_stream); } cuda_stream_handle_ = std::move(maybe_stream.value()); } } return GXF_SUCCESS; } gxf::Parameter<gxf::Handle<gxf::CudaStreamPool>> cuda_stream_pool_; bool default_stream_warning_ = false; std::vector<cudaEvent_t> cuda_events_; gxf::Handle<gxf::CudaStream> message_cuda_stream_handle_; gxf::Handle<gxf::CudaStream> cuda_stream_handle_; }; } // namespace holoscan } // namespace nvidia #endif/* GXF_EXTENSIONS_UTILS_CUDA_STREAM_HANDLER_HPP */
```