jpienaar Rename to underlying type rather than alias ... History

3 contributors

183 lines (164 sloc) | 7.5 KB ...

```
/* Copyright 2020 The TensorFlow Authors. All Rights Reserved.

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.
==============================================================================*/

#include "tensorflow/compiler/jit/xla_platform_info.h"

#include "tensorflow/compiler/xla/client/client_library.h"

namespace tensorflow {

xla::StatusOr<absl::optional<std::set<int>>> ParseVisibleDeviceList(
    absl::string_view visible_device_list) {
  std::set<int> gpu_ids;
  if (visible_device_list.empty()) {
    return {{absl::nullopt}};
  }
  const std::vector<string> visible_devices =
      absl::StrSplit(visible_device_list, ',');
```

```
  for (const string& platform_device_id_str : visible_devices) {
    int32_t platform_device_id;
    if (!absl::SimpleAtoi(platform_device_id_str, &platform_device_id)) {
      return errors::InvalidArgument(
          "Could not parse entry in 'visible_device_list': '",
          platform_device_id_str,
          "'. visible_device_list = ", visible_device_list);
    }
    gpu_ids.insert(platform_device_id);
  }
  return {{gpu_ids}};
}

Status BuildXlaCompilationCache(DeviceBase* device, FunctionLibraryRuntime* flr,
                                const XlaPlatformInfo& platform_info,
                                XlaCompilationCache** cache) {
  if (platform_info.xla_device_metadata()) {
    *cache = new XlaCompilationCache(
        platform_info.xla_device_metadata()->client(),
        platform_info.xla_device_metadata()->jit_device_type());
    return Status::OK();
  }

  auto platform =
      se::MultiPlatformManager::PlatformWithId(platform_info.platform_id());
  if (!platform.ok()) {
    return platform.status();
  }

  StatusOr<xla::Compiler*> compiler_for_platform =
      xla::Compiler::GetForPlatform(platform.ValueOrDie());
  if (!compiler_for_platform.ok()) {
    // In some rare cases (usually in unit tests with very small clusters) we
    // may end up transforming an XLA cluster with at least one GPU operation
    // (which would normally force the cluster to be compiled using XLA:GPU)
    // into an XLA cluster with no GPU operations (i.e. containing only CPU
    // operations).  Such a cluster can fail compilation (in way that
    // MarkForCompilation could not have detected) if the CPU JIT is not linked
    // in.
    //
    // So bail out of _XlaCompile in this case, and let the executor handle the
    // situation for us.
    const Status& status = compiler_for_platform.status();
    if (status.code() == error::NOT_FOUND) {
      return errors::Unimplemented("Could not find compiler for platform ",
                                   platform.ValueOrDie()->Name(), ": ",
                                   status.ToString());
    }
  }
```

```
79
80    xla::LocalClientOptions client_options;
81    client_options.set_platform(platform.ValueOrDie());
82    client_options.set_intra_op_parallelism_threads(
83        device->tensorflow_cpu_worker_threads()->num_threads);
84
85    string allowed_gpus =
86        flr->config_proto()->gpu_options().visible_device_list();
87    TF_ASSIGN_OR_RETURN(absl::optional<std::set<int>> gpu_ids,
88                        ParseVisibleDeviceList(allowed_gpus));
89    client_options.set_allowed_devices(gpu_ids);
90
91    auto client = xla::ClientLibrary::GetOrCreateLocalClient(client_options);
92    if (!client.ok()) {
93      return client.status();
94    }
95    const XlaOpRegistry::DeviceRegistration* registration;
96    if (!XlaOpRegistry::GetCompilationDevice(platform_info.device_type().type(),
97                                             &registration)) {
98      return errors::InvalidArgument("No JIT device registered for ",
99                                     platform_info.device_type().type());
100   }
101   *cache = new XlaCompilationCache(
102       client.ValueOrDie(), DeviceType(registration->compilation_device_name));
103   return Status::OK();
104 }
105
106 XlaPlatformInfo XlaPlatformInfoFromDevice(DeviceBase* device_base) {
107   auto device = static_cast<Device*>(device_base);
108   se::Platform::Id platform_id = nullptr;
109   const XlaDevice::Metadata* xla_device_metadata = nullptr;
110   std::shared_ptr<se::DeviceMemoryAllocator> custom_allocator;
111
112   if (device->device_type() == DEVICE_CPU) {
113     platform_id = se::host::kHostPlatformId;
114   } else if (device->device_type() == DEVICE_GPU) {
115     platform_id = device->tensorflow_gpu_device_info()
116                         ->stream->parent()
117                         ->platform()
118                         ->id();
119   } else if (XlaDevice::GetMetadataFromDevice(device, &xla_device_metadata)
120                  .ok()) {
121     // If we are on an XlaDevice, use the underlying XLA platform's allocator
122     // directly. We could use the StreamExecutor's allocator which may
123     // theoretically be more correct, but XLA returns a nice OOM message in a
124     // Status and StreamExecutor does not.
125     //
126     // Importantly we can't use ctx->device()->GetAllocator() as the allocator
127     // (which xla_allocator above uses) as on an XlaDevice, this is a dummy
```

```
    // allocator that returns XlaTensor objects. The XlaCompiler needs a real
    // allocator to allocate real buffers.
    platform_id = xla_device_metadata->platform()->id();
    custom_allocator =
        xla_device_metadata->client()->backend().shared_memory_allocator();
  }

  return XlaPlatformInfo(DeviceType(device->device_type()), platform_id,
                         xla_device_metadata, custom_allocator);
}

std::shared_ptr<se::DeviceMemoryAllocator> GetAllocator(
    DeviceBase* device, se::Stream* stream,
    const XlaPlatformInfo& platform_info) {
  if (platform_info.custom_allocator()) {
    return platform_info.custom_allocator();
  }
  auto* alloc = device->GetAllocator({});
  if (!stream) {
    // Stream is not set for the host platform.
    se::Platform* platform =
        se::MultiPlatformManager::PlatformWithId(platform_info.platform_id())
            .ValueOrDie();
    return std::make_shared<se::TfAllocatorAdapter>(alloc, platform);
  }
  return std::make_shared<se::TfAllocatorAdapter>(alloc, stream);
}

XlaCompiler::Options GenerateCompilerOptions(
    const XlaCompilationCache& cache,
    const FunctionLibraryRuntime& function_library, DeviceBase* device,
    se::Stream* stream, const XlaPlatformInfo& platform_info,
    bool has_ref_vars) {
  XlaCompiler::Options options;
  options.client = static_cast<xla::LocalClient*>(cache.client());
  if (stream != nullptr) {
    options.device_ordinal = stream->parent()->device_ordinal();
  }
  options.device_type = cache.device_type();
  options.flib_def = function_library.GetFunctionLibraryDefinition();
  options.graph_def_version = function_library.graph_def_version();
  options.allow_cpu_custom_calls =
      (platform_info.platform_id() == se::host::kHostPlatformId);
  options.device_allocator = GetAllocator(device, stream, platform_info);
  if (platform_info.xla_device_metadata()) {
    options.shape_representation_fn =
        platform_info.xla_device_metadata()->shape_representation_fn();
  }
  // If reference variables are not present in the graph, we can safely alias
```

```
  // passthrough parameters without performing a copy.
  options.alias_passthrough_params =
      !has_ref_vars && !platform_info.is_on_xla_device();
  return options;
}

}  // namespace tensorflow
```