Tensorflow XLA Service 详解 I

1181 Jiang XIAO

2019年9月26日 at pm1:24 (last edited 2020年6月20日 at am9:29)

compiler/aot/	以AOT的方式将tf2xla/接入TF引擎
compiler/jit/	以JIT的方式将tf2xla/接入TF引擎， 核心是9个优化器和3个tfop，其中XlaCompileOp调用tf2xla的“编译”入口完成功能封装，XlaRunOp调用xla/client完成“运行”功能。
compiler/tf2xla/	对上提供xla_compiler.cc:XlaCompiler::CompileFunction()供jit:compile_fn()使用将cluster转化为XlaComputation。核心是利用xla/client提供的接口，实现对XlaOpKernel的“Symbolic Execution”功能。每个XlaOpKernel子类均做的以下工作: **从XlaOpKernelContext中取出XlaExpression或XlaOp, 调用xla/client/xla_buidler.h提供的方法完成计算, 将计算结果的XlaOp存入XlaKernelContext.**
compiler/xla/client/	对上提供xla_builder.cc:Builder等供CompileFunction()使用，将Graph由Op表达转化为HloModuleProto:HloComputationProto:HloInstructionProto表达并保存在XlaComputation中。 对上提供local_client.cc:LocalClient::Compile()，作为编译入口供jit：BuildExecutable()使用，将已经得到的XlaComputation交给service并进一步编译为二进制。 对上提供local_client.cc:LocalExecutable::Run()，作为运行入口供jit/kernels/xla_ops.cc:XlaRunOp使用，通过Key找到相应的二进制交给service层处理
compiler/xla/service/	对上提供**local_service.cc:LocalService::BuildExecutable()**供LocalClient::Compile()使用实现真正的编译，承接XlaComputation封装的HloProto, 将其转化为HloModule:HloComputation:HloInstruction表达, 对其进行优化之后, 使用LLVM后端将其编译为相应Executable后端的二进制代码 对上提供**executable.cc:Executable::ExecuteOnStream()**供LocalExecutable::Run()使用实现真正的执行二进制。

编译cubin

调用栈：

```
e->tensorflow::XlaCompilationCache::BuildExecutable(entry->compilation_result, &entry
```

```
 2.     compile_result = xla::LocalClient::Compile()
 3.       executable = xla::LocalService::CompileExecutable()
 4.         execution_options = xla::CreateExecutionOptions()
 5.         xla::Service::CreateModuleConfig()
 6.         executor = execute_backend_->stream_executor()
 7.         xla::Service::BuildExecutable()
 8.           module = xla::gpu::CreateModuleFromProto()
 9.             module = HloModule::CreateFromProto()
10.             xla::gpu::NVPTXCompiler::RunHloPasses()
11.           executable = xla::gpu::NVPTXCompiler::RunBackend()
12.             llvm::Module llvm_module(module->name().c_str(), llvm_context);
13.             std::unique_ptr<StreamAssignment> stream_assignment = AssignStreams(*module
14.             hlo_schedule = GpuHloSchedule::Build()
15.             buffer_assignment = BufferAssigner::Run()
16.             IrEmitterUnnested ir_emitter();
17.             entry_computation->Accept(&ir_emitter)
18.             llvm_ir::DumpIrIfEnabled(*module, llvm_module, /*optimized=*/false);
19.             ptx = CompileToPtx(&llvm_module, {cc_major, cc_minor}, module->config(), li
20.               ptx =  CompileModuleToPtx()
21.                 target_machine = GetTargetMachine()
22.                 module_passes.add(...
23.                 module_passes.run(*module);
24.                 return EmitModuleToPTX(module, target_machine.get());
25.                   llvm::raw_string_ostream stream(ptx);
26.                   llvm::buffer_ostream pstream(stream);
27.                   codegen_passes.add(new llvm::TargetLibraryInfoWrapperPas)
28.                   target_machine->addPassesToEmitFile(codegen_passes, pstream)
29.                     codegen_passes.run() //dump_ir_pass.cc
30.                       for i in passes_.size():
31.                         llvm::legacy::PassManager::add(P);
32.                       llvm::legacy::PassManager::run(module);
33.             cubin = CompilePtxOrGetCachedResult(ptx, module->config())
34.               XLA_SCOPED_LOGGING_TIMER("NVPTXCompiler::CompilePtxOrGetCachedResult");
35.               if !ptx.empty():
36.                 //本质就是把string 的ptx变成uint8
37.                 StatusOr<std::vector<uint8>> maybe_cubin = se::cuda::CompilePtx(stream_
38.                   tensorflow::WriteStringToFile(env, ptx_path, ptx_contents);
39.                   std::vector<string> ptxas_args = {ptxas_path, ptx_path, "-o", cubin_p
40.                   ptxas_info_dumper.SetProgram(ptxas_path, ptxas_args);
41.                   tensorflow::ReadFileToString(tensorflow::Env::Default(),cubin_path, &
42.             module->entry_computation()->Accept(&cost_analysis);   --> 这个是啥，性能分析
43.             auto thunk_schedule = absl::make_unique<ThunkSchedule>(ir_emitter)
44.             gpu_executable = new GpuExecutable(cubin, thunk_schedule)
45.             return gpu_executable
46.       return new LocalExecutable(executable)
47.     executable = std::move(compile_result.ValueOrDie())
48.     out_compilation_result = &entry->compilation_result
```

-2- Client端Graph编译入口

-3- LocalService端Graph编译入口

-7- Service的Graph编译入口

-8- 根据Client端生成的HloProto表示的Graph转换为Hlo格式

-10- 优化HloModule, backend是XlaOp过滤逻辑用到的, 这里是service/gpu/nvptx_compiler.cc-11- 编译HloModule入口　nvptx_compiler.cc

-12- 构造最终提交到LLVM的llvm::Module对象

-13- 按照PostOrder的顺序依次给HloInstruction分配stream number, 根据是否是GEMM, 决定是否复用operand的

stream number.
-14- 分配Stream number, 决定了HloInstruction最终的处理顺序, 核心工作时确定了thunk_launch_order_以及据此构造的hlo_ordering_. 如果是配置为单Stream, 就是PostOrder, 否则使用BFSLaunchOrder, 会根据HloInstruction之间的依赖关系, 以及一共可用的Stream number数量来给每个HloInstruction分配Stream Number, 原则上, 会将不存在依赖关系的HloInstruction尽量分配到不同的Stream number, 如果有依赖, 那么这个HloInstruction的Stream number会和某个它所依赖的HloInstruction使用相同的Stream number, 所谓的launch order 并不是最终执行的顺序
-15- ****显存优化****
-16- visitor 也可以是别的visitor, 之前优化HloInstruction的时候就用到很多
-17- 遍历每一个HloInstruction 构造相应的Thunk, 除了全图的Accept, Instruction也有自己的Accept()用于局部遍历
-19- nvptx_backend_lib.cc 根据HloInstruction 生成ptx
-33- 根据ptx生成cubin, 即把string 的ptx变成uint8-39- 这个有没有优化空间
-42- **这个是性能分析???**

执行cubin

调用栈:

```
tensorflow::XlaRunOp::Compute()
  run_result = xla::LocalExecutable::Run() //friend class LocalClient
    return executable_->ExecuteOnStreamWrapper()
      return_value = ExecuteOnStream()     //GpuExecutable, gpu_executable.cc
        return Execute()
          globals = ResolveConstantGlobals(executor)
            if !cubin().empty():
              module_spec.AddCudaCubinInMemory(cubin());
              module_spec.AddCudaPtxInMemory(ptx().c_str());
              module_handles_.emplace()
          buffer_allocations = buffer_allocations_builder.Build()
          ExecuteThunks(buffer_allocations)
            se::Stream* main_stream = run_options->stream();
            se::StreamExecutor* executor = main_stream->parent();
            HloExecutionProfiler profiler()
            for thunk in thunk_schedule_->TotalOrder():
              thunk->Initialize(*this, executor);
                kernel = CreateKernel(executable.ptx(), executable.cubin()) //xla::gp
                  loader_spec.AddCudaPtxInMemory(ptx, kernel_name)
                  loader_spec.AddCudaCubinInMemory(cubin_data.data(), kernel_name)
                    cuda_cubin_in_memory_.reset(new CudaCubinInMemory{bytes, kernelna
                  stream_exec->GetKernel()
                kernel_cache_.emplace(executor, kernel)
              int32 stream_no = thunk_schedule_->StreamNumberForHlo(*thunk->hlo_instr
              se::Stream* stream =(stream_no == 0 ? main_stream : sub_streams[stream_
              thunk->ExecuteOnStream()          //xla::gpu::KernelThunk::ExecuteOnS
                it = kernel_cache_.find(executor)
                kernel = it->second.get()
                ExecuteKernelOnStream(*kernel)
                  for buf in args:
                    kernel_args->add_device_memory_argument(buf);
                  stream->parent()->Launch(kernel_args)
                    implementation_->Launch(kernel, args)
                      CUstream custream = AsGpuStreamValue(stream);
                      const GpuKernel* cuda_kernel = AsGpuKernel(&kernel);
                      CUfunction cufunc = cuda_kernel->AsGpuFunctionHandle();
                      void **kernel_params = const_cast<void **>(args.argument_addres
```

```
38.                         GpuDriver::LaunchKernel(context_, cufunc, custream, kernel_para
39.                           cuLaunchKernel()
40.             root = hlo_module_->entry_computation()->root_instruction()
41.             return std::move(shaped_buffer);
42.         return return_value
43.     launch_context.PopulateOutputs(ctx, run_result)
44.       output.set_buffer(se::OwningDeviceMemory(), {output_num});
45.       ctx->set_output(i, output_tensor);
```

-23- 将kernel加入kernel_cache, cache的作用是防止load 的时间占用了执行的时间, 让execute的统计更准确
-34- 获取custream
-39- 加载kernel到GPU执行, 注意, 加载时机和实际执行时机不是一回事, CPU端只需批量加载, GPU负责顺序执行stream上两个kernel

Related:
Tensorflow XLA Service Buffer优化详解
Tensorflow XLA Service 详解 II
Tensorflow XLA Service 详解 I
Tensorflow XLA Client | HloModuleProto 详解
Tensorflow XlaOpKernel | tf2xla 机制详解
Tensorflow JIT 技术详解
Tensorflow JIT/XLA UML
Tensorflow OpKernel机制详解
Tensorflow Op机制详解
Tensorflow Optimization机制详解
Tensorflow 图计算引擎概述

技术 Tensorflow, XLA, 技术

One comment on "Tensorflow XLA Service 详解 I"

Pingback: Tensorflow OptimizationPassRegistry机制详解 - sketch2sky

Leave a Reply